

Let's Talk Python



MEAP

Young coders build software

Pavel Anni

 MANNING



MEAP Edition
Manning Early Access Program
Let's Talk Python
Version 02

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing MEAP for *Let's Talk Python*.

This book is based on a real story. My son Erik liked to go to Starbucks and try drinks with different flavors and toppings. One day he decided to prepare drinks himself and treat his friends. He took his tablet to collect orders from them, but I suggested create a simple program for that. He tried to learn programming before but most of the exercises were boring for him. This time he saw a real problem he could solve with programming and he got interested. This is how this book was started. I hope you, dear reader, will find your own problem that can be solved with programming. And I hope this book will help you.

A couple pieces of advice.

Don't rush. I understand your desire to go directly to the last chapter, download the code of the final version of the program, and run it. Don't do it. Go step by step, write the code yourself (don't copy and paste, please!), try it, and move forward. Sometimes you will have to return and re-read the chapter. Sometimes you will need a break. Don't worry, take a break, repeat the chapter. Just don't drop out.

Make mistakes. You don't learn when everything goes perfectly well. The only way to learn is to make mistakes. Don't be afraid of mistakes. Experiment with the code, change things, get error messages, read them. Search for the error message on the Internet and discover thousands of other people who made the same mistake. Learn how they fix it and fix yours. Move ahead and don't drop out.

Ask questions. Ask your friends, ask parents and grandparents, ask Internet. Explain your problem to somebody – sometimes that's enough to find the answer yourself. There is no such thing as "stupid questions", don't be shy. Ask questions and don't drop out.

Go further. Modify the application you create with this book. Change something to make it look more like your own app. Think about other applications you can create. Look around you: what can be automated? Can you create an app that is similar to the app or website you know? Tell your friends about your ideas – maybe you will create something together? Programming is cool. Don't drop out.

Speak out. Please let me know your thoughts in the [liveBook Discussion forum](https://livebook.manning.com/#!/book/lets-talk-python/discussion) on what's been written so far and what you'd like to see in the rest of the book. Your feedback will be invaluable in improving *Let's Talk Python*.

—Pavel Anni

brief contents

- 1 Coffee for friends: first steps*
- 2 Lists: What's on the menu?*
- 3 Functions: Don't repeat yourself!*
- 4 User errors: Everybody makes mistakes*
- 5 Working with files: Being a shop manager*
- 6 Main menu: Next customer!*
- 7 Creating functions: Get the order and print it*
- 8 Working with JSON: Save the order*
- 9 Complete the menu: A real program*
- 10 Learning Flask: Your first web application*
- 11 Web form for orders: Coffee shop on the web*
- 12 Styles: Making it pretty*
- 13 Next steps: plans for the future*

APPENDIXES

- A Ideas for your first application*
- B How to install Mu Editor and Python environment*

Coffee for friends: first steps

This chapter covers

- Erik gets an idea
- Erik and Simon discuss the future application
- Erik installs a code editor and tries to run his first program in Python
- Simon explains how to use variables
- Erik writes his first dialogue in Python

1.1 A Great Idea

It all started on a sunny summer day. Erik came home with an idea: he wanted to prepare coffee drinks for his friends. Who knew that he would create his own online application for that?

"I will make it just like at Starbucks, with many flavors and toppings," he thought. "I think I have everything I need: coffee, three or four flavors to add, and some chocolate cream for toppings. Great!"

"Where is my iPad?" he asked his older brother Simon.

"Where you left it. Why?"

"I need it to collect orders for my coffee shop!"

He came back several minutes later with notes on his iPad, prepared four drinks for his friends, and left again.

"Wasn't it a good idea?" he asked Simon when he came home with four empty plastic cups.

"Yes, great idea," Simon said. "But..."

"What '*BUT*'??" Erik asked. He felt that his older brother wanted to ruin his day. As he usually did.

"You used your iPad to take orders, but you used it just as a plain paper notepad. You could create a simple application for your coffee shop and use it to take orders."

"You mean—like in an online shop? With menus and all that?" Erik already imagined *his own* web store with a huge title at the top: "Erik's Coffee Shop."

"Yes, of course. You know a bit of Python from that online course you've taken, don't you?"

"Yes, but I don't remember much. We did some exercises... I think it will be difficult—to make it look like a real online shop."

"Don't worry," Simon said. "We'll do it step by step. I did several projects like this for my robotics team at school."

NOTE

Don't worry if you didn't have any programming experience before. Erik didn't remember much from his classes anyway so we'll start from the very beginning.

Simon was in his last year in high school. He learned Python several years ago and used it in the school's Computer Science club and, more recently, in his Robotics team.

"So you are saying we can build a real online application?" Erik was not convinced.

"Yes, sure. If you don't drop out from my class," Simon smiled, "you will build it in a couple of weeks. Then, your customers will be able to choose whatever drink they want, add flavors..."

"And toppings!" Erik added.

"Yes, and toppings. And after they confirm the order, you'll see it on the orders page. And you will know what to prepare and for whom. Something like this," and Simon took a piece of paper and started to draw a simple web page.

"This will be your order page."

Erik's Coffee Shop

Enter your name:

Choose your drink

- ☐ Coffee
- ☐ Chocolate
- ☒ Decaf

Choose your flavor

- ☐ Caramel
- ☐ Marshmallow
- ☒ Strawberry

Choose your topping

- ☐ Chocolate
- ☐ Caramel
- ☒ Cinnamon
- ☐ Vanilla

Order

Cancel

"And this will be your list of orders."

Erik's Coffee Shop

Orders

Name	Product	Flavor	Topping
Alex	Coffee	Caramel	Vanilla
Drew	Decaf	Strawberry	Chocolate

"Cool! Do you think we can do it??" Erik still couldn't believe his brother.

"Of course! As I said: just don't drop out. You have plenty of time to finish it during your summer break."

NOTE

We have several other project ideas that you can use if you don't like the coffee shop idea. Some of them will be discussed when Erik's friends join him in the following chapters. Look for more details in the Appendix A.

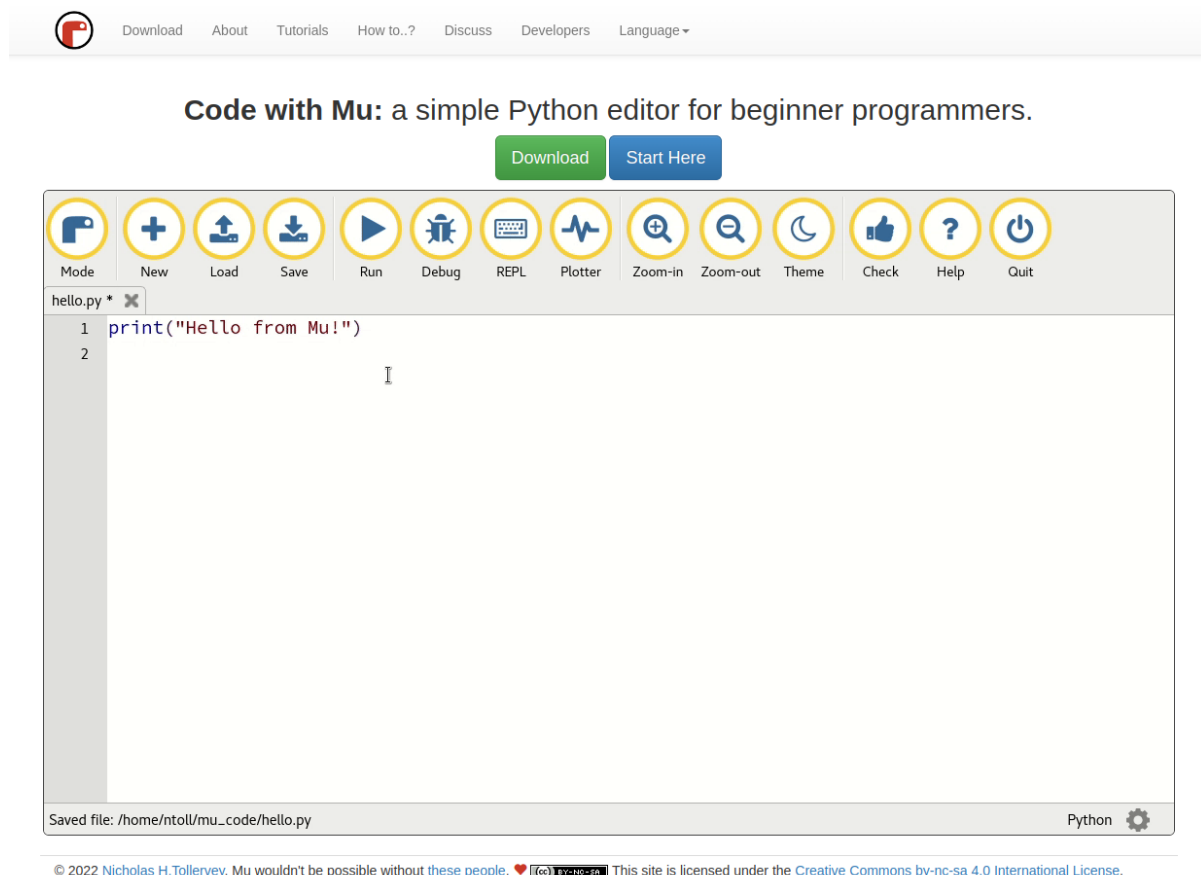
1.2 First things first: installation

"Let's start with some simple things. You will remember Python very quickly. Do you have it installed on your laptop?" Simon asked.

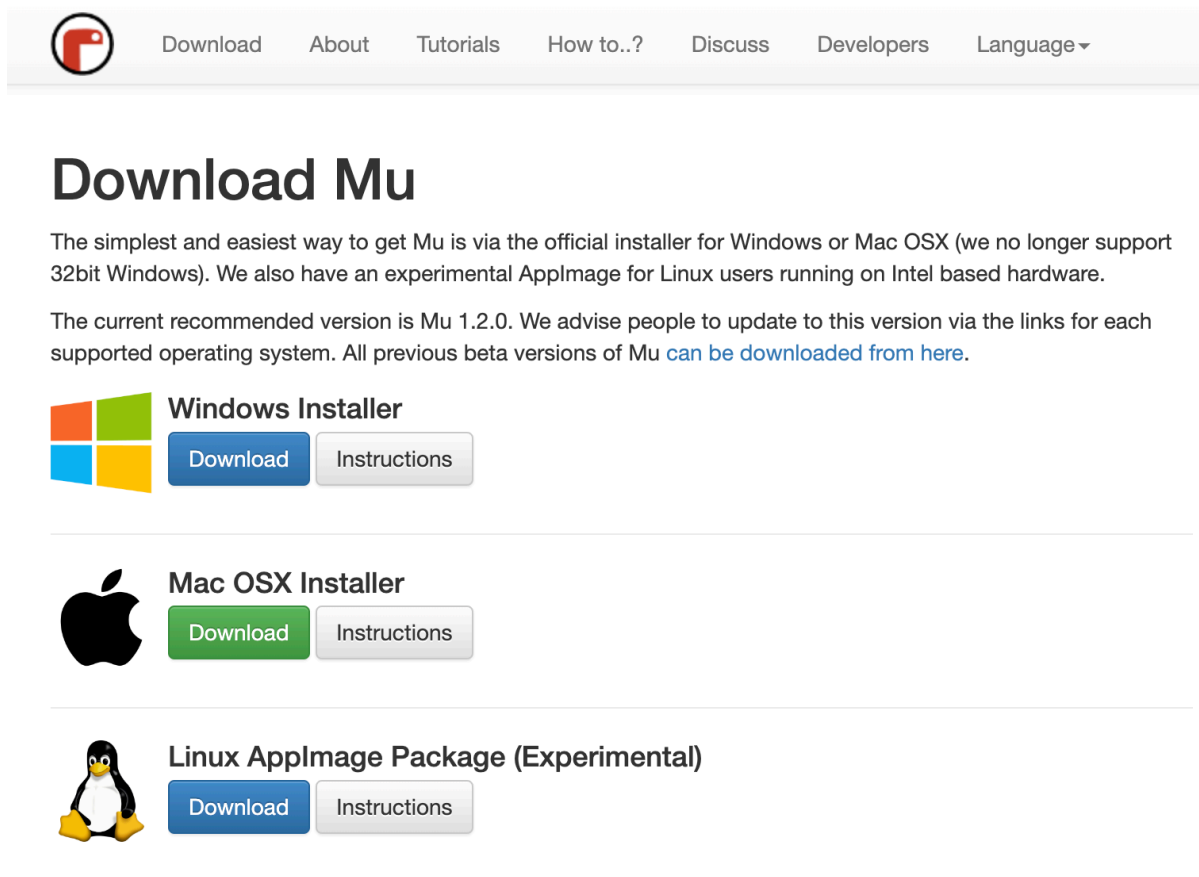
"No, I don't think so."

"Here is a great Python editor, designed specifically for beginners like you. It's called Mu Editor. Try to find it and install it. You can do it, I'm sure."

Erik found the website: <https://codewith.mu/>, where he could download it.



He downloaded the installation program from this page: <https://codewith.mu/en/download>.



Download Mu

The simplest and easiest way to get Mu is via the official installer for Windows or Mac OSX (we no longer support 32bit Windows). We also have an experimental AppImage for Linux users running on Intel based hardware.

The current recommended version is Mu 1.2.0. We advise people to update to this version via the links for each supported operating system. All previous beta versions of Mu [can be downloaded from here](#).

Windows Installer

[Download](#) [Instructions](#)

Mac OSX Installer

[Download](#) [Instructions](#)

Linux AppImage Package (Experimental)

[Download](#) [Instructions](#)

He clicked Instructions and found an instructions page with all the steps for his computer (at that time he was using a Macbook, but the site contains instructions for all three operating systems: Windows, macOS, and Linux.)

You can find all necessary links and instructions in Appendix B.

"Don't worry, it's not a toy. It's a perfect editor," Simon said. "We use it in our robotics team to work with microcontrollers. As you see, there are versions for Windows, macOS, and Linux. I use the Linux version in my team."

"Are there other editors for Python?" Erik didn't want just to follow his brother's directions.

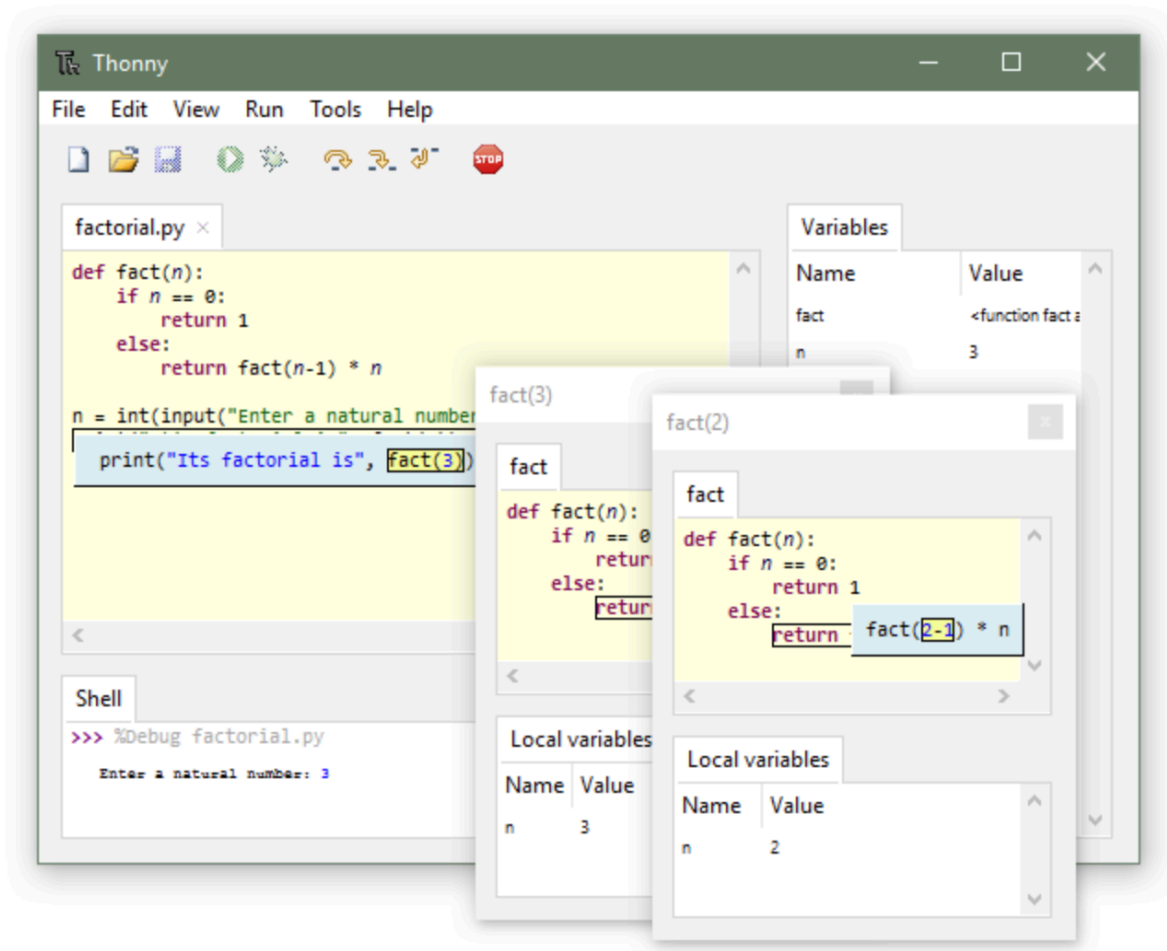
"Yes, of course, many of them. Another good option for beginners is Thonny. Look here: <https://thonny.org/> "

Thonny

Python IDE for beginners



Download version [4.0.1](#) for
Windows • Mac • Linux



"I like it!" said Erik. "And the name is funny."

"And, of course, there are other code editors that work on every platform:

- VS Code (<https://code.visualstudio.com/>),
- Sublime Text (<http://www.sublimetext.com/>).

"They all work perfectly with Python. Even the very old editors like Vim (<https://www.vim.org/>) and Emacs (<https://www.gnu.org/software/emacs/>) support Python, but you have to be a *very serious* programmer to use them," and Simon winked at his brother.

"Mu Editor and Thonny," Simon continued, "both *include* Python when you install them. To use Python with *other* editors, you have to *install* it first. On some systems, like Linux and macOS, Python is already installed from the beginning. On Windows, you should install it. I can show

you later if you want."

YOUR TURN Install your Python environment

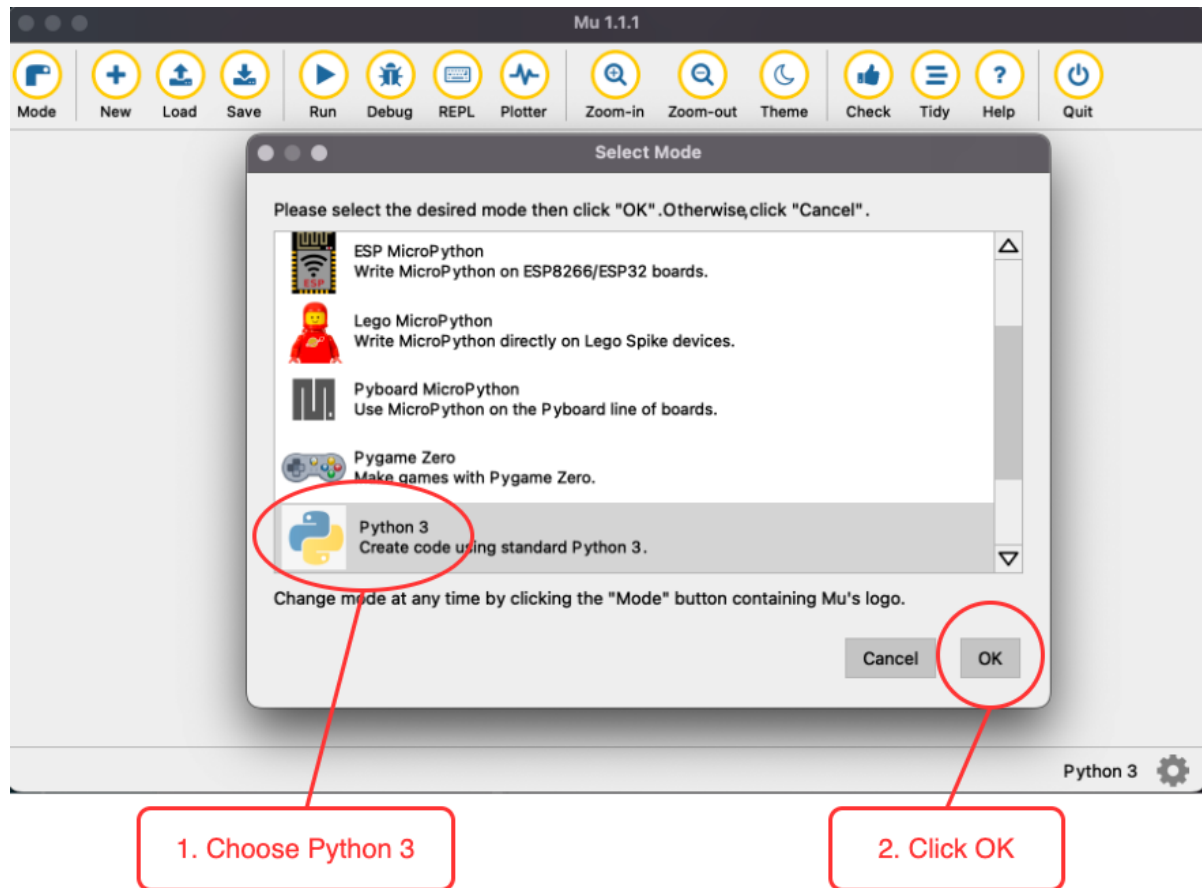
Now it's your turn. Open your laptop or desktop, and install Mu Editor. You can find the complete instructions for different platforms in Appendix B (it is available on Windows, macOS, Linux).

If you prefer some other editor, feel free to install it instead of Mu. Don't be afraid to experiment!

1.3 How to talk to a computer

"Let's start Mu Editor and begin writing your coffee shop program," Simon said.

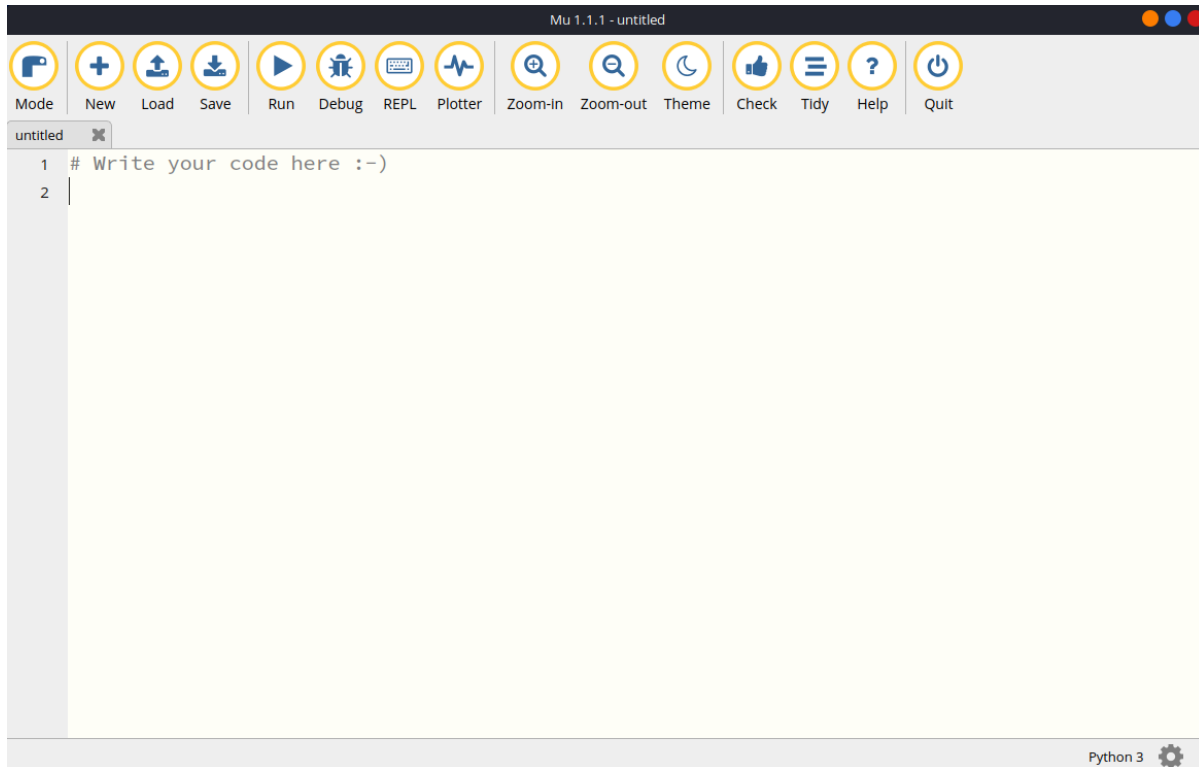
Erik launched Mu and saw its first window:



"Select Python 3 from the menu and click OK," Simon suggested.

Erik did what Simon said. "From now on," Simon continued, "Mu Editor will remember that you prefer to use Python 3. Maybe you noticed that there are some other modes that can be used to work with microcontrollers, build web applications, and others. We will learn about them later."

Now Erik had the editor window in front of him.



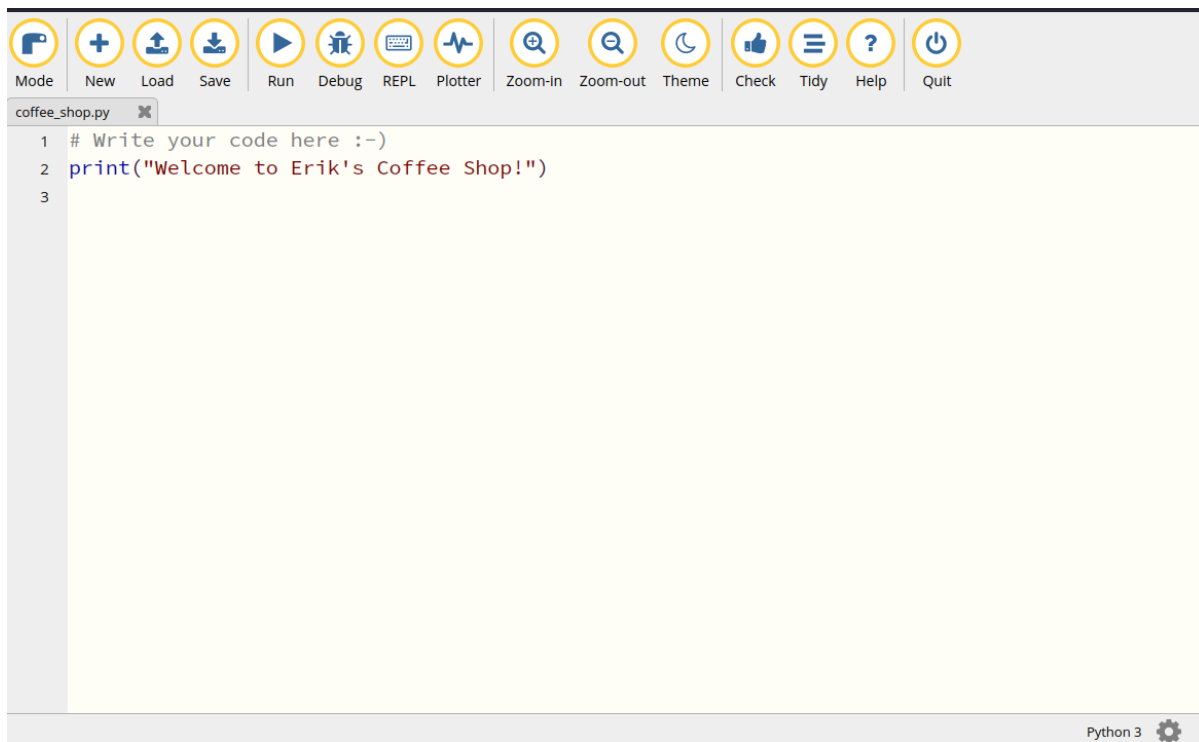
"What should I write here?" Erik asked.

"What do you want your program to do first?"

"It should say 'Welcome to Erik's Coffee Shop!'"

"Great! Let's write it. Remember the function `print()` in Python?"

Erik started to write. This first step was easy.



"Now what?"

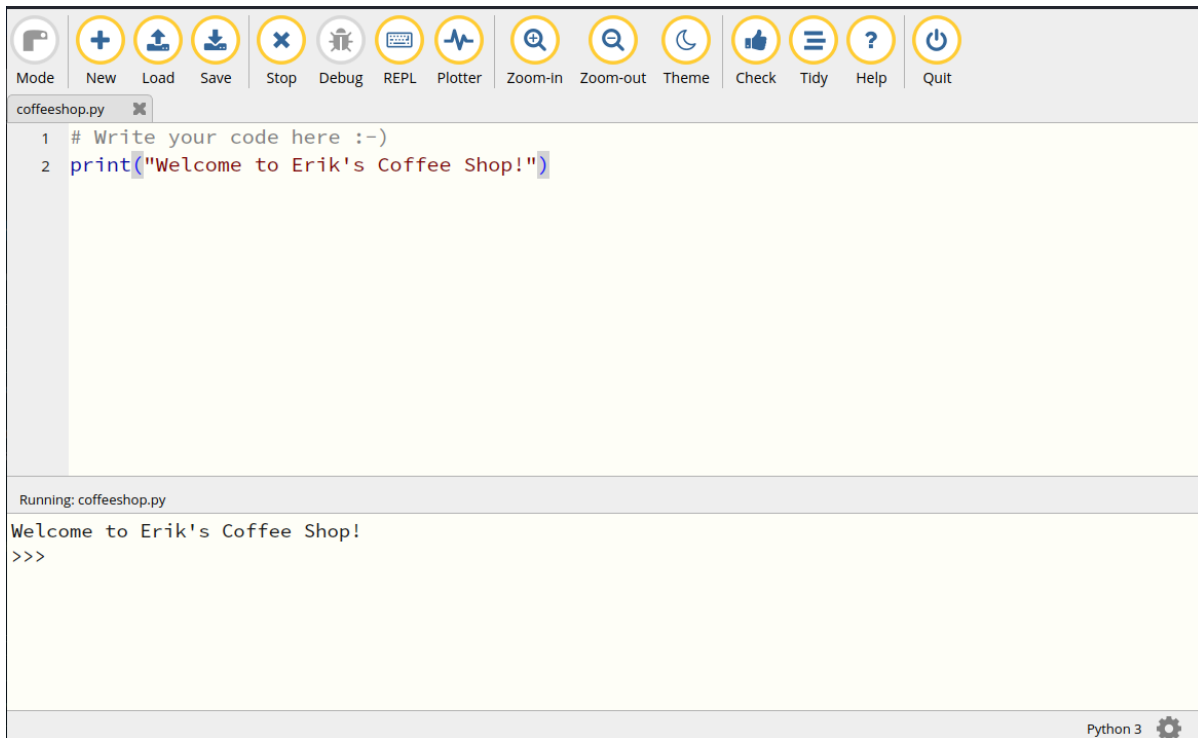
"Now you run it. Click Run  Run ."



Erik clicked the button, and suddenly another window appeared asking if he wanted to save the program. That was easy. Erik typed the name of the file: "coffeeshop" and was ready to press ENTER to save the file when Simon said:

"Wait, wait... Don't forget to add `.py` to the file name. You have to let your text editor know that it's a Python program. Mu Editor will add it automatically, but other editors won't. So make sure all your Python files are named with `.py` at the end."

Erik added `.py` to the file name and saved the file. Immediately after, he noticed another window at the bottom of the editor's window. There was the coffee shop greeting—precisely as he wanted it!



"It works!" Erik was delighted.

"Of course, it works. Why shouldn't it?" Simon answered. "But you wanted to collect orders, didn't you?"

"Yes, I would ask my client's name and what they want..."

"And then?..." Simon obviously knew the answer, but he wanted Erik to find it himself.

"And then I would print 'Hello! Here is your order:' and show their name, flavor and topping. Like on a real receipt."

"Good idea," Simon said. "But look: when you are writing your program, you don't know what your friend wants to order, right? So you can't write in your program 'You ordered caramel.' Also, different clients order different things. It will be caramel for Alex and strawberry for Emily. So you see: your flavor *varies* from order to order, as well as the client's name. Remember what this thing is called in programming?"

"It's a *variable*!" Erik was glad he remembered it from the Python course he took several months ago.

"Right!" Simon was glad too. "Variable is like a box: you can put something into it, and then open and see what's in the box. You can replace what's in the box with something else."

"In our case," Simon continued, "let's start with a box called 'answer' and store whatever you hear from your client in that box. You ask your client their name and they answer 'Alex,' for

example. You put this answer in the box called 'answer' and keep it there. When you want to print it out, you tell Python: 'please print whatever is now in the box called 'answer'. The next client's name is Emily, and now you put 'Emily' in the box. And next time, Python will print 'Emily' not 'Alex' because it is what is *now* in the box called 'answer.' Let's write the code for this."

"Right here, in the same file?" Erik asked.

"Sure, go ahead and continue in the same file. To get something from the client, we use the *function* called `input()`. When you call it, it waits for the user to enter something. So the user types something on the keyboard and presses `ENTER`. And then the function *returns* whatever the user entered."

"Wait, wait," Erik stopped Simon. "What does it mean--'returns'? And also, you are talking about *functions*. Of course, I know what they are, but can you tell me what *you* mean by 'functions'?" Erik didn't want to show that he *barely* remembered something about functions from his previous class.

"A function is a piece of code that *does* something. Almost *any* piece of code does something, but some pieces of code we use more often than the others. Later, you will create your own functions, but for now, we will use the functions written by somebody else. There are operations that people use very often, such as print something. You didn't notice it, but you already used a function when you wrote `print()` in your previous program. In programming we say that you *call* a function."

"A-ha, I see," Erik said. "Something with parentheses is called a 'function'."

"Right. And you can put something inside those parentheses, and the function will *do* something with it. For example, it will print your message. What you pass into a function is called *arguments*. Sometimes it's a string, sometimes it's a number, sometimes there are several arguments."

"We call it 'to *pass* arguments' to a function," Simon continued. "The function will do something with the arguments and get something as a *result*. For example, it can calculate something, or do something with the string that you passed, like converting it to ALL CAPS or encrypting it. And then it *returns* that result to your main program."

"But how do I see the result?" Erik asked. "Will the function print it?"

"No, it won't. Here is where we need *variables*. We tell Python: '*please call this function with these arguments and please put whatever it returns into this box, sorry, this variable*'. And all that is done using a simple 'equal' sign, like this `=`. For example, if you want to call the function `input()` and put what it returns to the variable `answer`, you simply write:"


```
answer = input()
```


"And after you save the client's answer, you can print it. You call the `print()` function and pass your variable as an argument."

"Great," Erik said, "now I see how to write it." He started writing in the editor. In a minute or two, he's got this:

```
print("Welcome to Erik's Coffee Shop!")  
  
answer = input()  
print(answer)
```

"Should I run it?" he asked Simon.

"Sure, go ahead, click [Run]  Run ."

Erik clicked [Run]  Run .

"It says 'Welcome to Erik's coffee shop' and then nothing."

"What did you expect?" Simon asked.

"That it will ask me my name."

"But you didn't tell Python that it should ask something. Now it's waiting for your input. Type something."

Erik typed: "Erik" and pressed `ENTER`.

Python printed: "Erik".

The screenshot shows the Mu Python IDE window titled "Mu 1.1.1 - coffeeshop.py". The top toolbar includes icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, Zoom-out, Theme, Check, Tidy, Help, and Quit. The main editor area contains the following Python code:

```
1 # Write your code here :-)
2 print("Welcome to Erik's Coffee Shop!")
3
4 answer = input()
5 print(answer)
```

Below the editor, the output console shows the program's execution:

```
Running: coffeeshop.py
Welcome to Erik's Coffee Shop!
Erik
Erik
>>>
```

The bottom status bar indicates "Python 3" with a settings gear icon.

"It works!" Erik said.

YOUR TURN Write your first dialogue

Write the dialogue program that Erik just wrote. It's a short program, we recommend to type it yourself instead of copying from the book. Create a name for your coffee shop and use it the first "Welcome" message. You can create some other shop if you want. What will it sell? Ice cream? Flowers? Pet toys?

Try to run your program. Does it do what you expect it to do? If it doesn't, copy it from the book or from our web site: <https://github.com/pavelanni/lets-talk-python-book> and run it again. It should work.





"Yes, it works," Simon said, "but let's make it more user-friendly. Remember, you were confused when it said nothing except 'Welcome'? You should tell your user what you expect from them. And also, instead of printing just 'Erik', you could add something like 'Here is your order, Erik'."

"You can pass this string to the `input()` function as an argument. We call it a *prompt* string. It explains what we expect from the user. And in the `print()` function, you can add the string you want to print before the `answer` variable. Let me help you."

Simon helped Erik to add those strings to the code, and this is what it looked like after that:

```
print("Welcome to Erik's Coffee Shop!")

answer = input("Please enter your name: ")
print("Here is your order, ", answer)
```

Simon noticed that Erik was looking for the [Run]  Run button and explained: "Before clicking [Run]  Run again, you have to stop your previous Python session. See these three angle brackets here? They mean that Python is running and waiting for your input. We will use it later, but for now, just click [Stop]  Run and then click [Run]  Run again."

Now the program asked for the order and answered exactly as Erik programmed it.

```
Welcome to Erik's Coffee Shop!
Please enter your name: Erik
Here is your order, Erik
>>>
```

YOUR TURN Make your program more user-friendly

Add the prompt and the output string to your first program. Create a different prompt asking the client their name, like "Glad to see you! What's your name? "

"Looks much more user-friendly, doesn't it?" Simon said. "Always think about your users and ask yourself: Is it clear enough what I expect from the user? Can they possibly make a mistake here?"

"Now," Simon continued, "we have to ask your client about their order. You said you have coffee and chocolate. And also you said something about flavors and toppings?"

"Yes," Erik said, "I want to ask them which topping and flavor they want."

"Well, go ahead and ask them. You can just repeat the same code—but don't forget to change the prompts. And I think you should print the whole order at the end, not after each question. Try it."

Erik wrote this code and stopped at the last line.

```
print("Welcome to Erik's Coffee Shop!")

answer = input("Please enter your name: ")
answer = input("Please enter your drink: ")
answer = input("Please enter your flavor: ")
answer = input("Please enter your topping: ")
print("Here is your order: ", answer)
```

"You told me to put the answers in the `answer` variable. But how do I know now which is the flavor and the topping?" Erik was confused.

"Yes, I told you to put the answers in a variable, *for example called 'answer'*," Simon answered. Here we come to one of the most difficult problems in computer science: naming variables", he smiled. "Of course, you don't store all the answers in the variable called `answer`. Let's use different variables for different answers and give them meaningful names. For the client's name we'll use a variable called `name`--that's easy. If you ask about a main drink, put the answer in the variable `drink` or `product`. For the flavor and topping answers use the variables `flavor` and `topping`."

"At the end," Simon continued, "print each variable on a separate line, using several `print()` functions. Go ahead. I will help you if necessary."

Erik worked on his code and finally produced this:

Listing 1.1 coffeeshop.py

```
print("Welcome to Erik's Coffee Shop!")

name = input("Please enter your name: ")
drink = input("Please enter your drink: ")
flavor = input("Please enter your flavor: ")
topping = input("Please enter your topping: ")
print("Here is your order, ", name)
print("Main product: ", drink)
print("Flavor: ", flavor)
print("Topping: ", topping)
print("Thanks for your order!")
```

Erik clicked Run, and his program started a dialogue. Erik answered all the questions and got a nice output:

```
Welcome to Erik's Coffee Shop!
Please enter your name: Erik
Please enter your drink: coffee
Please enter your flavor: caramel
Please enter your topping: chocolate
Here is your order, Erik
Main product: coffee
Flavor: caramel
Topping: chocolate
Thanks for your order!
>>>
```

Simon noticed the last line and praised Erik for his initiative: "It's always good to thank your customers."

"Yes, I saw that on several receipts in coffee shops," Erik said. He was glad he had done something on his own besides what his older brother told him.

YOUR TURN Add more options to the dialogue. Use variables.

Edit your previous program and add the other lines to the dialogue. Again, feel free to change the prompts and strings you print to something more suitable to your project.

Change your printed output. Look at the receipts from the places you visit (coffee shops, restaurants, groceries, other shops). Try to make your printed output look similar. Use text symbols like `|`, `_`, `=`, `+`, and others to make your output look interesting.

Simon decided it was time to wrap up for today.

"I think it was a good start today," he said. "Let's see what we have done today. First, we installed your programming environment."

"Yes," Erik said. "I like this Mu Editor. It uses colors to show me different parts of the program. And also it shows my string in red until I put the quotes at the end. And it has a dark mode! I know that *real programmers* always use the dark mode! I think I will continue using it."

"Second," Simon said, "we used a *function* for the first time. What was it?"

"It was the `print()` function," Erik said. "I told it what to print and it printed it."

"Right. You *called* the function and *passed an argument* to it."

"Third," Simon continued, "you used another function to get information from the user."

"It was `input()`," said Erik. "And I saved the answers in *variables*."

"Great!" Simon was really proud of his brother. "You are making good progress."

YOUR TURN Explain it yourself

Try to explain it in your own words.

- What is a function? Give some examples.
- What are function arguments? Give some examples.
- How do you call a function?
- What is a variable? Why do we need them?
- How should we name our variables? Why?

1.4 What is a program?

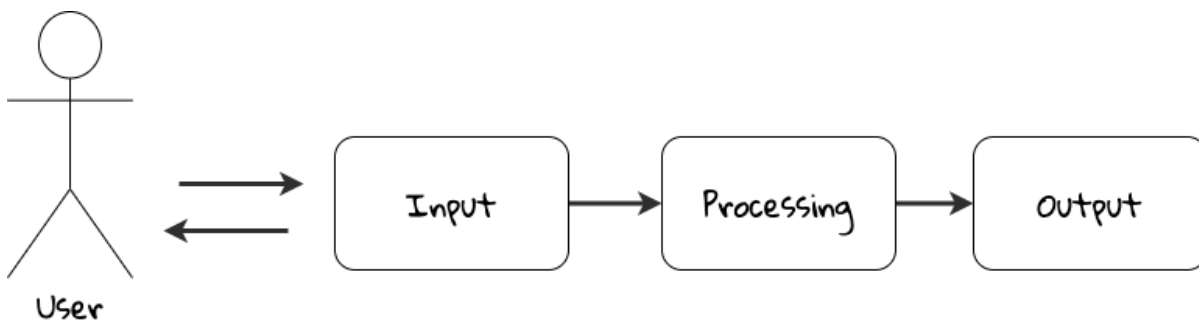
"Finally, let's add a bit more theory," Simon said. "We just built a very simple program. But it has all the main components of any other program. We have asked the user for some *input*. After we received data from the user, we did something with that data. We usually call it *processing*. In our case, we just stored that data, but we could do something else with it, right?"

"For example?" Erik asked.

"For example, you've entered 'coffee' in lower case, but we may want to start all products and flavors from a capital letter. There is a special function in Python for that. So we can *process* the data after we received it."

"Great idea, I want to add it!" Erik said.

"Sure, we'll do it. And finally, after we processed the data, we printed it out. In other words, we produced some *output*. Look here."



"Input doesn't always come from a user," Simon continued. "Sometimes there is no interaction with a user, and the program takes data from somewhere else. For example, from the Internet, like recent sports results. Sometimes from sensors, like in robotics. Or from documents and images."

"Very often the output is not just a print output. In my robotics team we get inputs from sensors, we process them in the microcontroller, and our output is the signals to motors like: 'turn left, move forward.' But the structure is still the same: input → processing → output."

"Enough theory," Simon said. "Tomorrow we'll work on improving your program."

"Improving?" Erik was surprised. "But it works fine already, doesn't it?"

"What if your user enters something you don't have in your shop?" Simon asked. "Like 'maple syrup'? What will you do? You should tell your user what you have in your coffee shop and what they can order. So, tomorrow we'll work on *menus*. And also, we'll see what we can do in case of errors."

1.5 New terms we have learned today

- **Variable**
a place (a box) where we can store some values. For example, we can store numbers, letters, strings. A variable can hold only one thing at a time.
- **Function**
a piece of code that does something and that we want to be able to do it again.
- **Function arguments**
information that the function needs to do its job.
- **To call a function**
to write the name of the function with parenthesis and arguments between them.

1.6 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch01>

Lists: What's on the menu?

This chapter covers

- Erik starts using menus in his coffee shop
- Erik uses Python lists to keep his drinks, flavors, and toppings
- Erik uses for-loops to print out his lists
- Simon explains how computers store numbers and strings
- Erik learns an important thing about list indexes

Next day Erik was ready to continue working on his Coffee Shop application. He remembered that Simon said something about missing products that customers might enter in the dialogue. He came to his brother and asked:

"You said yesterday that customers can enter something that I don't have in the shop. What should I do about it?"

"Remember the last time you were in a coffee shop or restaurant. How did you know what you can order?"

"They had a menu with a list of products that they have in this shop."

"Right!" Simon said. "A menu! This is what we are going to create today. How does a menu look like in a coffee shop?"

"It's a list. A list of main drinks like coffee, chocolate, decaf. And a list of flavors I can add. Like caramel, mint, and others. And a list of toppings."

"Right, lists!" Simon was very glad the Erik used that word. "Like this, right?" and he quickly drafted something that looked like a menu.

Menu

Drinks

Coffee	1.00
Chocolate	1.50
Decaf	1.20

Flavors

Caramel	0.50
Vanilla	0.45

...

Toppings

Chocolate	0.30
Cinnamon	0.30

....

"Lists is what we need! We have lists in Python—you may remember that. Lists are very useful in Python. They can contain numbers, strings, even other lists. For example," Simon took another piece of paper and wrote several examples.

```
fruits = ['apple', 'peach', 'banana']
```

```
numbers = [42, 256, 1000]
```

```
constants = [3.1416, 2.718, 1.4142]
```

Let's create lists for your menu. You just give the list a name—like `flavors`, for example, and then list your flavors in square brackets. And the same for your toppings and main drinks. Don't forget that your flavors, toppings, drinks are strings so they should be in quotes. You can start a new file in your editor and call it `menu.py`, for example."

Erik opened his editor and started writing. Here is what he's got in several minutes.

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]
```

"Very good," Simon said. "Now let's print them as menus."

"Just `print(drinks)`?" Erik suggested.

"You can do that, but it won't be pretty. Try it."

Erik added the `print()` statement at the end.

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]
print(drinks)
```

"Click Run  Run," Simon said.

Erik clicked and saw the output at the bottom of the editor window.

```
['chocolate', 'coffee', 'decaf']
>>>
```

"If your menu is a list, we should print it as a list," Simon said. "And also you should give your user a way to choose from the list. For example, you can ask to type a letter. But here we have chocolate and coffee. So you can't use the letter 'C' for both. Let's use numbers instead. For each menu item we'll print a number. Then your user will type a number for their choice. For example, 1 for chocolate, 2 for coffee. Something like this." Simon took a piece of paper and drew a simple menu.

Erik's Coffee Shop

Choose your drink

1 chocolate

2 coffee

3 decaf

Type the number:

Choose your flavor

1 caramel

2 vanilla

3 peppermint

Type the number:

"Yes, I saw that in a Chinese restaurant—each dish had a number," Erik remembered. "But how do I do it in Python?"

"You have a list of several items," Simon started his explanation. "You have to print each item adding a number in front of it. When we have to *repeat* something in Python we use a *loop*. In this case it will be a *for-loop*. You tell Python that for each item in the list it has to do something. Like print it, for example."

"Write a simple for-loop," Simon continued. "Let me write the first one for you." Simon took Erik's keyboard and added a couple of lines to his code.

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

for d in drinks:
    print(d)
```

Simon clicked [Run] and they saw the output:

```
chocolate
coffee
decaf
>>>
```

"Now it's your turn," Simon said. "Write the same code for the other two lists. Note that I used

another variable to print list items. The list is called `drinks`. I used just a `d` for each drink in the list. It is usually recommended to use meaningful names for your variables, like `drinks` for the list of drinks. But if a variable will be used just in one loop to go through a list, it can be short, one or two letters. It's not a rule, but it's easier to type."

"Another important thing," Simon continued, "is that in Python spaces mean *a lot*. You see that the `print()` function call is shifted four spaces to the right? This is how we tell Python *what* should be repeated in the loop. The part that is shifted is called a *block*. Everything you put in this block will be repeated for each list item. Now there is only one function call, but we'll add something later.

"You also noticed that I didn't type four spaces on the keyboard. Our editor did it for us automatically. All programming editors that you would use for Python have this feature. When they see the colon (`:`) they automatically shift the next line. It is called 'starting a block'. Now go ahead and write the loops."

Erik created two more loops to print the other two lists. He liked the idea to use shorter variable names (less typing!). He also noticed that the editor shifted the line after colon automatically. Very useful! Here is what he's got.

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

for d in drinks:
    print(d)

for f in flavors:
    print(f)

for t in toppings:
    print(t)
```

He saved the program and ran it.

YOUR TURN Create your list of products and print it

Open your code editor and create a program similar to what Erik just created. It should contain three or more lists of items. Then use loops to print out the content of those lists.

You can use Erik's menu item or you can create your own. Ice cream flavors, bagels, minifigures, anything you want!

```

chocolate
coffee
decaf
caramel
vanilla
peppermint
raspberry
plain
chocolate
cinnamon
caramel
>>>

```

"Very good," Simon said. "But we don't have the numbers. We have to fix it. Remember I told you that we can add something else to the block? Here is what I propose. We'll create a variable which will keep the item's number in the list. Each time we go to the next item we add one to that variable. In that case 'chocolate' will be number one, 'coffee'—number two, and so on."

"Let me show you," and Simon took Erik's keyboard again.

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

i = 1
for d in drinks:
    print(i, d)
    i = i + 1

for f in flavors:
    print(f)

for t in toppings:
    print(t)

```

He ran the program and they saw this:

```

1 chocolate
2 coffee
3 decaf
caramel
vanilla
peppermint
raspberry
plain
chocolate
cinnamon
caramel
>>>

```

"You see: I added the `i` variable. For each list item now I print not only its value, but also its number. And then I add one to the number to move from 1 to 2, then from 2 to 3, and so on. Now go ahead and change the rest," Simon said.

Erik made the changes:

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

i = 1
for d in drinks:
    print(i, d)
    i = i + 1

for f in flavors:
    print(i, f)
    i = i + 1

for t in toppings:
    print(i, t)
    i = i + 1

```

When he ran the program he saw this output:

```

1 chocolate
2 coffee
3 decaf
4 caramel
5 vanilla
6 peppermint
7 raspberry
8 plain
9 chocolate
10 cinnamon
11 caramel
>>>

```

"But this is not what I wanted!" Erik said. "I think it should be: one, two, three for the drinks, then one, two, three for the flavors, and one, two three for the toppings again."

"Right!" Simon agreed. "How would you do this?"

"Use a different variable?"

"Yes, that's possible too. But you can use the same `i` variable. The important thing is to set it to one before each loop. We call it to *initialize* the variable."

Erik added `i = 1` before each loop and got this:

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

i = 1
for d in drinks:
    print(i, d)
    i = i + 1

i = 1
for f in flavors:
    print(i, f)
    i = i + 1

i = 1
for t in toppings:
    print(i, t)
    i = i + 1
```

YOUR TURN Print three (or more) menus with numbers

Modify your previous program to add numbers to your menu items. Use the loops. Don't forget to reset the item counter with each new list.

He clicked Run and got the output:

```
1 chocolate
2 coffee
3 decaf
1 caramel
2 vanilla
3 peppermint
4 raspberry
5 plain
1 chocolate
2 cinnamon
3 caramel
>>>
```

"Now let's make it a bit prettier," Simon said. "Add titles like 'Our drinks' before each list. Remember, we should let the user know what they see and what they should do."

Erik added the titles. He even added an extra line under each title. He was sure it will make it look like a real menu.

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1

print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1

```

YOUR TURN Add titles to your menus

Add titles to your menus to make the output beautiful. Use your shop's name in the titles. Try to use other symbols instead of dashes.

And the output was beautiful, as he expected:

```

Erik's Coffee Shop drinks
-----
1 chocolate
2 coffee
3 decaf
Erik's Coffee Shop flavors
-----
1 caramel
2 vanilla
3 peppermint
4 raspberry
5 plain
Erik's Coffee Shop toppings
-----
1 chocolate
2 cinnamon
3 caramel
>>>

```

"Looks good," Simon said. "What is also good about this format is that now you have three lists in your menu and three lists in your program."

"Let's write this code," Simon continued. "For each list in the menu, you have to ask the user to choose an item and get that information from them. How do you get information from a user? You did it yesterday, remember?"

"With `input()`?" asked Erik.

"Of course!" Simon was glad Erik remembered the previous lesson. "You can write it yourself, can't you?"

"Let me try," Erik said and started editing his code. He remembered that he should use the `input()` function. Then he put the prompt inside the parentheses and on the left side he used a variable. He remembered that he shouldn't use the same variable for different questions.

Here is what he wrote:

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1
drink = input("Choose your drink: ")

print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1
flavor = input("Choose your flavor: ")

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1
topping = input("Choose your topping: ")
```

YOUR TURN Add user inputs to your menus

Add the `input()` functions to your menus. Use appropriate variable names to store the user's answers.

"Now what?" he asked Simon.

"Now your user types a number and you use that number to find the item. In Python we call this number a list *index*. If you put this number in square brackets next to the list name, you get that item. Like this," and he wrote an example:

`drinks[drink]`

"So after you know the number you can find the item in the list. And you can print 'Here is your order' like you did yesterday, but now you'll take those items from the menu. Try it, I'll help you if necessary."

That was a bit more difficult. Erik looked at his yesterday's program and copied the lines from it to the bottom of this program. Then he replaced variables like `drink` with the list items like Simon suggested.

Here is his code:

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1
drink = input("Choose your drink: ")

print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1
flavor = input("Choose your flavor: ")

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1
topping = input("Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drinks[drink])
print("Flavor: ", flavors[flavor])
print("Topping: ", toppings[topping])
print("Thanks for your order!")
```

"Now run it and let's see what it gives us," Simon said.

Erik clicked [Run] and the program printed the drinks menu and asked for his choice. So far it worked. Erik quickly entered numbers for all three menus and saw this output:

```
Here is your order:
Traceback (most recent call last):
  File "/home/erik/mu_code/menu.py", line 30, in <module>
    print("Main product: ", drinks[drink])
TypeError: list indices must be integers or slices, not str
>>>
```

"What's this?" he was puzzled.

"Congratulations!" Simon said.

"What are you so happy about? That my program doesn't work?" Erik started getting angry at his brother.

"Not at all!" Simon said. "You got your first error message from Python and it's a good sign! Making errors and fixing them is the only way to learn. You've got an error message—now let's try to fix the problem. Usually Python gives you the reason why this happened. Start from reading the last message. What does it say?"

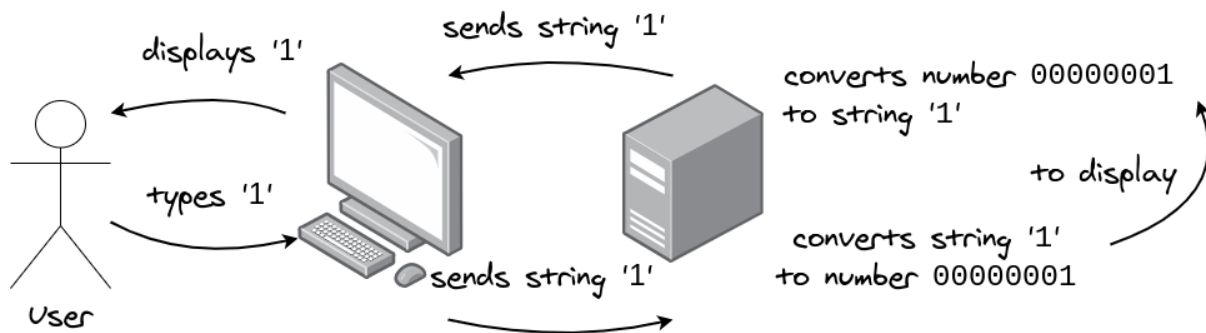
"Something about slices... Must be integers not str. What's that?"

"I agree, it takes some practice to learn to read Python error messages. Here it tells you that when you use a variable as an index of a list, that variable should be an integer number, like one, two, three."

"But I entered numbers!" Erik was still confused.

"Yes, you *typed* numbers on your keyboard. But for Python everything you enter from a keyboard is a *string*. Python makes a difference between a string containing the number '1' and the integer number 1."

"This comes from the way computers keep things in memory," Simon continued. "The computer keeps the *number* 1 in memory, but when it shows it to you, it converts it to a *string* '1'. The same way computers convert numbers they get from the keyboard. You type '123' on your keyboard and the computer gets this string and *converts* it to a *number* 123. Look here," and Simon draw a picture with a computer, keyboard, display, and a user.



"So we should tell Python to convert the strings you type on the keyboard to integer numbers. There is a special function for that called `int()`. Let me show how to use it."

Simon changed Erik's program in one place and let him do the same in the other two places. Here is what Erik's program looked like after that:

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1
drink = input("Choose your drink: ")

print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1
flavor = input("Choose your flavor: ")

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1
topping = input("Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drinks[int(drink)])
print("Flavor: ", flavors[int(flavor)])
print("Topping: ", toppings[int(topping)])
print("Thanks for your order!")

```

Erik ran the program, entered his choices (coffee, caramel, chocolate) and got this output:

```

Here is your order:
Main product: decaf
Flavor: vanilla
Topping: cinnamon
Thanks for your order!
>>>

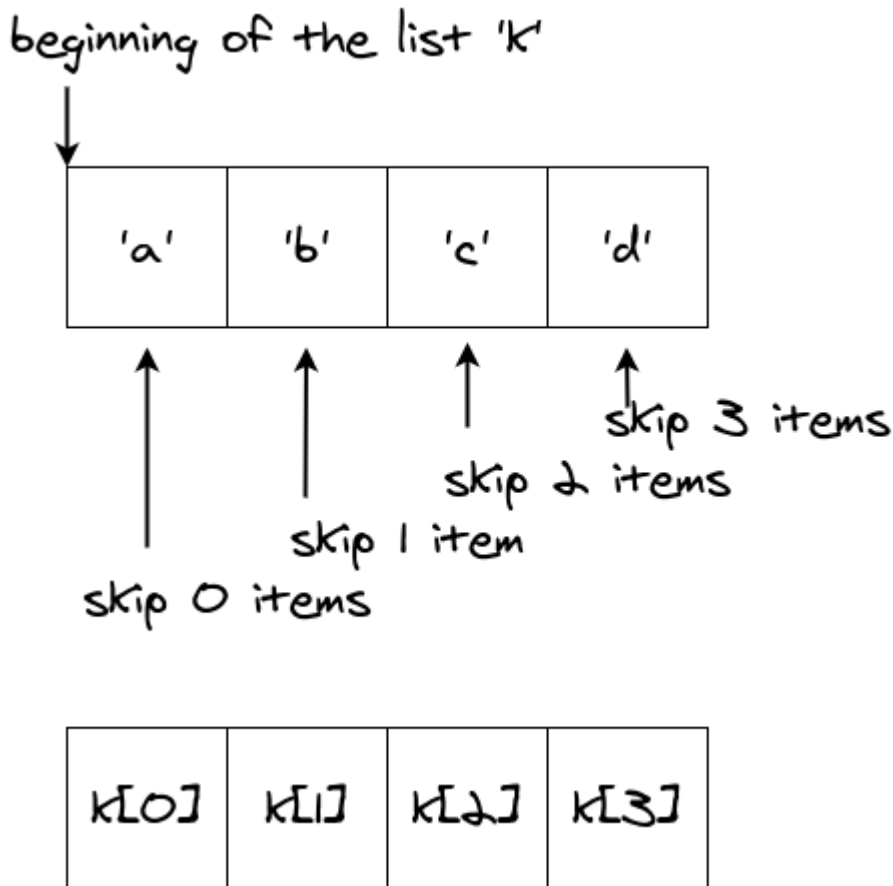
```

"What?? It's all wrong!" Erik exclaimed. "This is not what I chose!"

"I didn't tell you one important thing about Python lists. Their indices start with zero, not one. So if you want to get the first item from the list, you should put zero in the square brackets. If you want the second item, you use one as the index."

"But why??" Erik was shocked by such a strange thing.

"It's a long story," Simon answered. "It comes from the way computers store lists in memory. The index you use is the number of items you should skip from the beginning of the list to get the item you want. If you want the first item of the list you don't have to skip any items. You just take it from the beginning of the list. So the number of items you should skip is *zero*, right? That's why the first element's index is zero. Look here," and Simon draw another picture.



"So what should I do now?" Erik asked. He thought that he understood Simon's explanation, but still he was annoyed by this inconvenience.

"I see that you're annoyed," Simon said. "Don't worry, you'll get used to it very quickly. And you will, like all real programmers, start counting everything from zero," Simon smiled. "Now you just subtract one from each index in the square brackets. But be careful: you have to add that - 1 *after* you converted your input to integer, not before. Like this: `drinks[int(drink) - 1]`."

Erik fixed his code and now it looked like this:

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1
drink = input("Choose your drink: ")

print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1
flavor = input("Choose your flavor: ")

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1
topping = input("Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drinks[int(drink) - 1])
print("Flavor: ", flavors[int(flavor) - 1])
print("Topping: ", toppings[int(topping) - 1])
print("Thanks for your order!")

```

YOUR TURN Modify your program to print the order

Modify your program similar to what Erik just did and print the order. Don't forget to convert the input strings to numbers. Don't forget to subtract one (1) from each number—list indexes start with zero, remember?

He ran the program, entered 2, 1, 1 and finally got what he wanted:

```

Here is your order:
Main product:  chocolate
Flavor:  caramel
Topping:  chocolate
Thanks for your order!
>>>

```

"Cool! It works!" Erik was definitely happy. "I like my coffee shop program! Are we done with it?"

"Almost," Simon answered. "Look, you wrote almost exactly the same code three times."

"What's wrong with that?"

"Imagine you want to change something in your code. For example, change the way you print the menu items. You'll have to change it in all three places. Or in even more places if you decide to

add other menu lists. Imagine you want to add desserts to your coffee shop. That means you'll have to copy this code one more time. What if you made a mistake in the code? Programmers call them *bugs* (I'll tell you later why). Then you would have to fix that bug in four places, repeating yourself. Programmers like the DRY principle: Don't Repeat Yourself."

"But I don't see how I can do that," Erik was confused. "If I have *three* menu lists, I have to print them three times. And I have to ask the user for input three times."

"We can use a function here," Simon explained. "Remember when we started using the `print()` function, I told you that for operations that we want to repeat over and over we use functions. So far we used functions written by somebody else. Now we'll create our own function and use it."

"This is cool, I like it," Erik said.

"Great, let's do it tomorrow. I think we have done enough for today. You did a great job, Erik," Simon indeed was glad that his brother is making progress. "Let's recap what we have learned today. What was the first thing?"

"First, we created *lists*," Erik said. "We put all our drinks, flavors, and toppings in the lists."

"Good, what was next?"

"Then we printed the lists using loops. And then we printed the numbers next to each drink or flavor."

"Yes, exactly," Simon confirmed. "Go on, what was after that?"

"And then I tried to print drinks from the list, but I got an error from Python. And then you explained to me how numbers are stored in computer memory. Then we converted the numbers and I tried to print my order again. And *just because you didn't tell me that indices start with zero*," (Erik didn't forget that!) "my order was printed all wrong items!"

"Please, forgive me," Simon smiled. "But now you'll remember it much better, I'm sure!"

"Finally I fixed it and now it works well!" Erik finished.

"Great job!" Simon gave Erik the thumbs up. "We'll continue tomorrow and write our first function."

2.1 New things you have learned today

- *List*
a collection of items in Python. You can have strings or numbers in a list, or even a mix of them.
- *List index*
the number we can use to retrieve an item from a list. Indices in lists always start with zero and increase by one for each next element: 0, 1, 2, 3, etc.
- *Numbers and strings*
these are different *types* of variables in Python. When you print something on the screen, or get input from the keyboard, you always use strings. When you want to do any math operations with numbers you received from the user you have to convert them from strings to numbers.

2.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch02>

3

Functions: Don't repeat yourself!

This chapter covers

- Simon explains how to avoid repeating yourself (and why)
- Erik writes his first Python function
- Erik starts using Python's interactive tool
- Erik improves his function to make his receipt look professional

"Where did we stop yesterday?" Simon asked Erik on the next day.

"You said that I should not repeat myself. And also you said we are going to write our own function today."

"Right! First, tell me what you know about functions so far."

"We used a couple functions already," Erik started to answer. "We used `print()` and `input()`. You said that somebody has written them so we can use them. We can use arguments with functions. We just have to put them between the parentheses and the function will do something with the arguments, like print them."

"Everything is right! You are a great student!" Simon smiled. "The important thing about functions is that they can do *the same thing* but with different arguments. So if you see that you are doing the same thing several times you should look at whether it can be turned into a function. To decide you should look at your repeating code and ask yourself which parts are the same and which are different. Where is your script from yesterday?"

"Here," Erik opened the `menu.py` file in the editor.

```

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

print("Erik's Coffee Shop drinks")
print("-----")
i = 1
for d in drinks:
    print(i, d)
    i = i + 1
drink = input("Choose your drink: ")

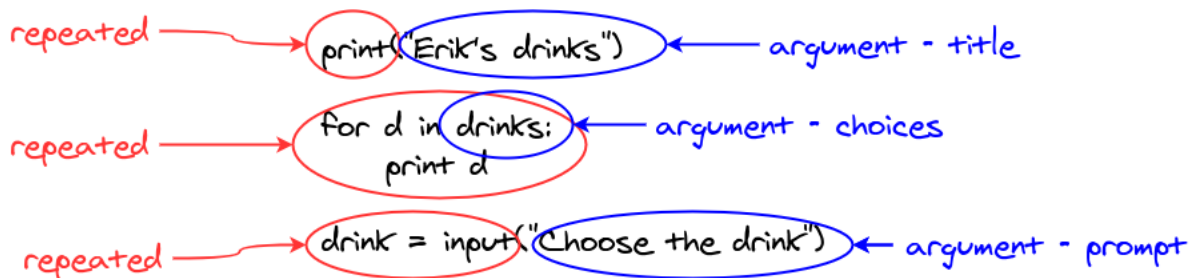
print("Erik's Coffee Shop flavors")
print("-----")
i = 1
for f in flavors:
    print(i, f)
    i = i + 1
flavor = input("Choose your flavor: ")

print("Erik's Coffee Shop toppings")
print("-----")
i = 1
for t in toppings:
    print(i, t)
    i = i + 1
topping = input("Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drinks[int(drink) - 1])
print("Flavor: ", flavors[int(flavor) - 1])
print("Topping: ", toppings[int(topping) - 1])
print("Thanks for your order!")

```

"Look: what is repeating here?" Simon asked and started drawing a diagram with Erik's code.



"The loop. The `print()` in the beginning and the `input()` in the end."

"And what is different in these three cases?"

"The prompt for the `input()` is different," Erik answered. "Also the title is slightly different."

"And also you are running for-loops through different lists, like drinks, flavors, and toppings, right?" Simon decided to help Erik. "So we will *pass* as arguments those things that are different. In our case they will be the list of choices, the menu title, and the input prompt."

"Let's start a new file in the editor, call it `menu_function.py`, and write our function there."

Erik clicked [New] in the editor, then [Save], typed `menu_function.py`, and was ready to write code.

"Functions in Python start with the word `def` followed by the name of the function," Simon continued. "Let's call our function `menu`. Then you open the parentheses and list your arguments."

Erik wrote `def menu(` and wasn't sure what to do next.

Simon helped: "We just talked about your arguments. I see you are thinking how to name them. Remember, naming variables and arguments is one of the most difficult problems in computer science? You are not alone. Let's name them: `choices`, `title`, and `prompt`. Just type them between the parentheses and put a colon after the closing one."

Erik typed the following:

```
def menu(choices, title, prompt):
```

He noticed that after he pressed `ENTER`, the cursor moved to the next line, but four spaces to the right. "Should I write here?" he asked Simon.

"Yes, sure!" Simon answered. "You see: the editor is helping you to write your function! Now look at your yesterday code and start copying what you want to put in the function. Look: first we print the title. Let's do it here too, but instead of the actual string we just print the *argument* called `title`. You can even put the line of dashes after it, like you did before."

Erik wrote:

```
def menu(choices, title, prompt):
    print(title)
    print("-----")
```

"Now write the loop," Simon continued. "But instead of drinks or flavors, your list is now called `choices`. And you can use the variable `c` in the loop, as the first letter of 'choice'."

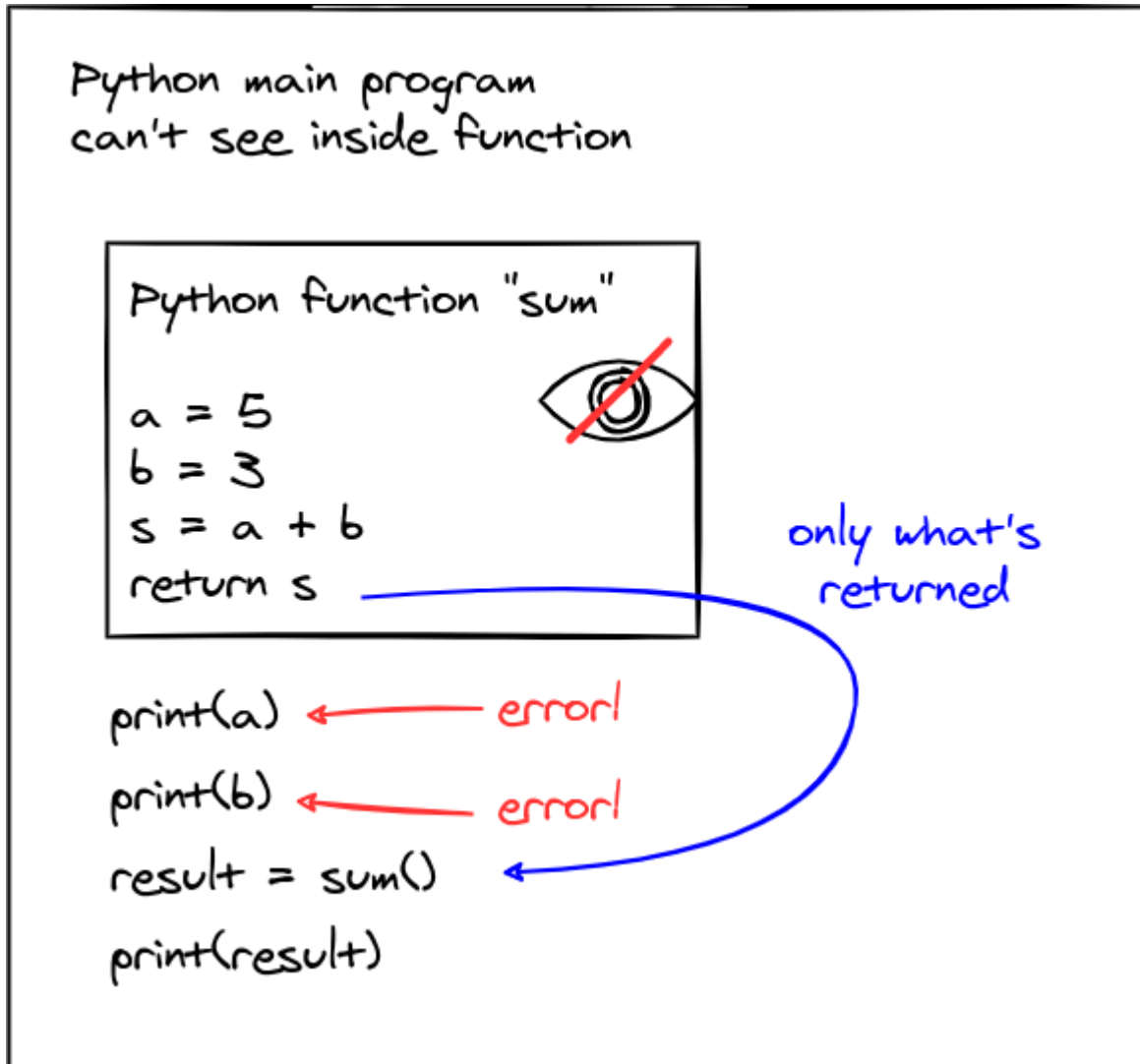
Erik copied the loop from his previous script. He got the idea now and added the `input()` function with the prompt even without asking his brother.

```
def menu(choices, title, prompt):
    print(title)
    print("-----")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)
```

"Great!" Simon said when he saw Erik's code. "Now we have to *return* the choice."

"Can't we just print the `choice` variable in our main program?" Erik asked.

"No! And this is a very important thing about functions," Simon was glad that Erik asked this question. "The variables you have inside your function are *visible* only within the function. Look here, I'll draw a picture."



Visible means that you can't see what's inside those variables when you are not *inside the function*. So if we want our main program to see their values we have to *return* these variables. Usually we have many variables inside a function, but we want to return only one or two as a result. In this case we can return the number that the user entered, which is stored in the variable `choice`."

Simon paused and thought for a moment.

"But we can do better," he said. "Look at your yesterday code again. What else is repeating?"

Erik looked and said: "Those `int()` functions and also that we had to add `- 1` three times. That

was annoying," he still hasn't gotten used to the fact that list indices start with zero.

"Okay, let's add them to the function too," Simon suggested. "We will convert the user's answer to integer, get the item from the list, and return the item, not its index. That will make our function even more useful. The main program that calls it will get the user's choice, not just some number. Let me show you," and Simon added the conversion operations and the `return` statement to the function.

```
def menu(choices, title, prompt):
    print(title)
    print("-----")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)
    answer = choices[int(choice) - 1]

    return answer
```

"Now let's test it," Simon said. "In our main program that goes just below the function we will call it and print the answer we get. But first, we need our lists with drinks and everything. Copy them from the top of your yesterday's program."

Erik added three lists just below the function. This time all the lines were not shifted and began in the first position.

```
drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]
```

"Good," Simon said. "Now you are ready to call your function. Pass the title, the list of drinks, and the input prompt. The result from the function assign to a variable. Call it `choice`, for example. And then print it."

Erik followed Simon's instructions and here is what he's got:

```
def menu(choices, title, prompt):
    print(title)
    print("-----")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)
    answer = choices[int(choice) - 1]

    return answer

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]
choice = menu(drinks, "Erik's drinks", "Choose your drink: ")
print(choice)
```

YOUR TURN Create your own function

Create your own function similar to what Erik just created. Use your menu lists, titles, and prompts. Try to run it (before Erik!).

"Should I run it?" Erik asked.

"Yes, it's now ready, go ahead!"

Erik clicked [Run] and the program asked him about his drink, exactly the same way it did before. Erik answered and got the result he expected.

```
Erik's drinks
-----
1 chocolate
2 coffee
3 decaf
Choose your drink: 2
coffee
>>>
```

"It works!" he said. "I'll add the other menus here," and he started writing. In 10 minutes or less he was ready to test the whole program. Now it looked like this:

```
def menu(choices, title, prompt):
    print(title)
    print("-----")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)
    answer = choices[int(choice) - 1]

    return answer

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

drink = menu(drinks, "Erik's drinks", "Choose your drink: ")
flavor = menu(flavors, "Erik's flavors", "Choose your flavor: ")
topping = menu(toppings, "Erik's toppings", "Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drink)
print("Flavor: ", flavor)
print("Topping: ", topping)
print("Thanks for your order!")
```

YOUR TURN Add other menus

Add other menus to your program. They will use the same function but with different arguments: lists of choices, titles, and prompts. Try to run it and test with your menu choices.

And it worked as expected! Erik ran the script, entered 2, 2, 1 again and got his order:

```
Here is your order:
Main product: coffee
Flavor: vanilla
Topping: chocolate
Thanks for your order!
>>>
```

Simon said: "Notice that your program became shorter. And now if you have to change something, you change it only in one place."

"Why would I want to change it? It works well already," Erik said.

"Oh, there are always ways to improve your code!" Simon answered. "Let's make its title a little bit better. Did you notice that your line of dashes is now longer than the title?"

"Really? Oh, yes, you are right," Erik said. "It's because I changed the title to just 'Erik's drinks'. I can make it shorter, that's easy."

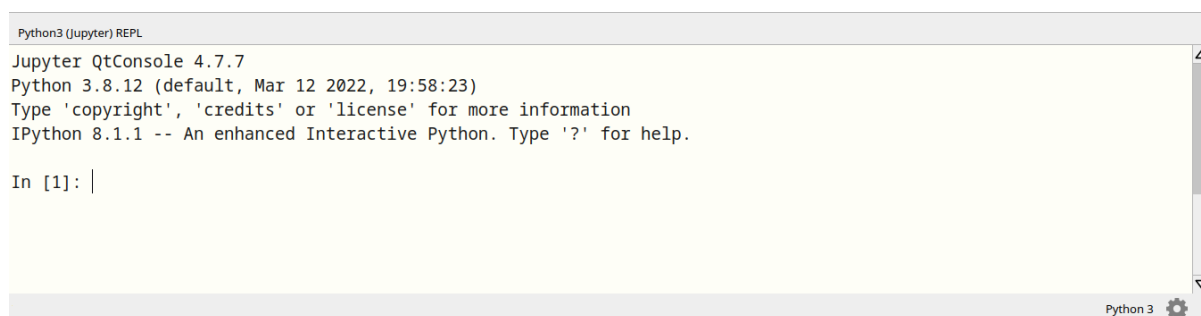
"Sure, you can make it shorter, but look—in the 'drinks' menu the title is shorter, in the 'toppings' menu it's longer. You have to print the line of dashes that will be the same length as your title string."

"But how can I do it? Maybe, it's better to remove that line of dashes?" Erik was confused.

"No, I like your line here; it makes your receipt look more real. I want you to keep it. But we have to calculate the length of our title and make it the same length. We'll learn one new function and one new operation here. The function we are going to use is called `len()`. You just put a string argument inside the parentheses and it returns the length of the string. Let me show you. We'll use another feature of Mu Editor, called **REPL**. It stands for 'Read-Eval-Print-Loop' and means that you can use Python interactively. I usually use it when I want to test something quickly. Or to show something to somebody, like now," and Simon smiled.

"Click [REPL]  REPL," Simon continued.

Erik clicked and another window opened at the bottom of the Mu Editor window.



```
Python3 (Jupyter) REPL
Jupyter QtConsole 4.7.7
Python 3.8.12 (default, Mar 12 2022, 19:58:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.1.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: |
```

"You see—you have interactive Python here," Simon said. "You can type any Python code here and it will be executed. You can even use it as a calculator," Simon smiled. "Try to type `print('hello world')` or `2 + 2` and see."

Erik typed and Simon was right indeed!

```
In [1]: print('hello world')
hello world

In [2]: 2 + 2
Out[2]: 4
```

"Now let's calculate the length of a string. Type: `len('abcd')`."

Erik typed and got this:

```
In [3]: len('abcd')
Out[3]: 4
```

"Now you see that the length of the string 'abcd' is 4," Simon said. "You can do the same with string variables too. Use the variable `s`, put the string 'hello' into it and calculate its length. I'm sure now you know how to do it."

Erik typed in the REPL window and got the result:

```
In [4]: s = 'hello'

In [5]: len(s)
Out[5]: 5
```

"Good," Simon said. "Now you know that if you have a string you can always get its length. Even more, you can get the length of a list this way. Create a list of numbers: 1, 2, 3, and get its length. Call it `n`, for example."

Erik typed:

```
In [6]: n = [1, 2, 3]

In [7]: len(n)
Out[7]: 3
```

"We will use it later, but now let me show you one trick," Simon continued. "What will Python give me if I ask it to take a number 2 and multiply it by 2?"

"4?" Erik wasn't sure if it's a trick already. The question was too simple.

"Right. What will Python give me if I take a letter 'A' and multiply it by 2?"

"I don't know? 2A, maybe?"

"Go ahead and try it with interactive Python!" Simon suggested.

Erik typed and got the result:

```
In [8]: 2 * 'A'
Out[8]: 'AA'
```

"Interesting!" Erik was surprised.

"Now what if you take a dash instead of 'A' and multiply by 10?"

Erik started to guess where Simon is leading him to and typed:

```
In [9]: 10 * '-'
Out[9]: '-----'
```

"And now replace the number 10 with the length of the string 'hello'."

Erik got Simon's idea now and typed:

```
In [10]: len('hello') * '-'
Out[10]: '-----'
```

YOUR TURN Use REPL and experiment with the `len()` function

Start REPL by clicking its icon in the editor. Repeat all Erik's experiments.

Try to multiply a number to a string of two or three letters. Can you guess what will be the output?

"I see it now!" he said. "We take the `title` argument, we calculate its length, and we print the line of dashes of exactly the same size!"

"Can you change your function now?" Simon asked.

"Yes, sure, I know what to do!" Erik started typing already. He changed only the third line (look for the label **1**) and now his function looked like this:

```
def menu(choices, title, prompt):
    print(title)
    print(len(title) * '-')
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1

    choice = input(prompt)
    answer = choices[int(choice) - 1]

    return answer
```

YOUR TURN Change your function to print the correct line of dashes

Make a change in your program to print the correct number of dashes, similar to what Erik just did. Try to use a different symbol (equal sign = or underscore _ or something else).

He tested the main program and now all lines of dashes were exactly of the same size that their titles.

Simon commented, "Now you see that not only the result that your function returns is dependent on the arguments, but even what it prints can be dependent as well."

"It's always a good idea to analyze the arguments you receive in your function," Simon continued. "In this case we checked the title's length. What would your function do if it received an empty string with length zero?"

"I don't know," Erik answered. "I think it will print an empty string, nothing."

"Right," Simon said. "But maybe we can still print something reasonable, even if the title is empty. Maybe just a word 'Menu' and a line of dashes. For such cases in Python we have *default* values for function arguments. In your function I would change the first line to this," and Simon edited Erik's file:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
```

"In this case we tell Python, 'If there *is* an argument `title` then accept its value. If you didn't use it when you called your function, then use its default value, which is just 'Menu'. It is usually recommended to set default values. You can always change them to something else when you call your function."

"Let's test it," Simon suggested. "In your first call with `drinks` remove both the title and the prompt. Leave only `drinks` as a single argument."

Erik did what Simon suggested and now the first function call looked like this:

```
drink = menu(drinks)
```

He ran the program again and saw the first menu:

```
Erik's Menu
-----
1 chocolate
2 coffee
3 decaf
Choose your item:
```

Here is Erik's full program.

Listing 3.1 menu_function.py

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)
    answer = choices[int(choice) - 1]

    return answer

drinks = ["chocolate", "coffee", "decaf"]
flavors = ["caramel", "vanilla", "peppermint", "raspberry", "plain"]
toppings = ["chocolate", "cinnamon", "caramel"]

drink = menu(drinks)
flavor = menu(flavors, "Erik's flavors", "Choose your flavor: ")
topping = menu(toppings, "Erik's toppings", "Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drink)
print("Flavor: ", flavor)
print("Topping: ", topping)
print("Thanks for your order!")
```

YOUR TURN Use default arguments in your function
Add default values for the title and prompt arguments. Try to call your function without those arguments and make sure it uses the default values.

"Of course, it's not telling the user that it's a *drinks* menu, but still it's better than just an empty string. It's helpful when you want to test something quickly. You can always add more descriptive titles and prompts later."

"I think that's enough for today," Simon said. "Let's recap what we have learned. What was the first thing today?"

"We looked at my program I wrote yesterday and found things that were repeated three times. And you told me that we can write our own function. You told me about the word `def` and the arguments."

"Good, go ahead," Simon encouraged Erik. "What about those arguments?"

"I used the list of choices, the title, and the prompt as arguments in my function." Erik liked to talk about *his* function—he wrote it himself for the first time!

"Then you showed me that **REPL** thing in the editor," Erik continued. "I like it! And then we calculated the string's length."

"And what did we use it for?" Simon asked.

"Yes, we used it to print our receipts and now they look beautiful. And then we tried to use default values for arguments. It was a bit boring but it worked."

Simon said: "This is a very important thing that you just said. Very often good programming solutions look boring. But they work. Programming is not always about fancy tricks and hacks. Most of the time you have to do very boring things, like check user's input, check for errors, and so on. But if doing this boring stuff makes your program work—it's worth it. Tomorrow we'll see what we have to do to make sure your program works even if your user enters wrong values. But for now—take some rest! You did a great job today!"

3.1 New things you have learned today

- *Function*
A piece of programming code that can be used (called) repeatedly. A function can be written by you or somebody else. If it's written by somebody else then usually it's part of a *library* or a *module* in Python.
- *Arguments*
Variables that we pass to the function when we call it. The function takes the arguments and uses them to prepare its output. The output can be printed or *returned* to the main program.
- *REPL*
Read-Eval-Print-Loop, a way to run Python interactively. Very useful to test some functions quickly.

3.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch03>

4

User errors: Everybody makes mistakes

This chapter covers

- Erik discovers that users don't always do what you tell them to do
- Erik learns about using loops to repeat his question to the user
- Simon helps Erik to make the menus more robust

"Yesterday you did a great job, Erik," Simon continued the next day. "You wrote a very good function, you added default arguments, you tested it."

"Yes," Erik answered. "I think it's a good program. I want to show it to my friends!"

"Wait, wait," Simon said. "I don't think it's ready to use yet."

"Why? I think it works perfectly!"

"Oh, really? Let me try," Simon looked like he had something in mind. He started Erik's program again and at the first menu he entered: **"coffee."**

```
Erik's drinks
-----
1 chocolate
2 coffee
3 decaf
Choose your drink: coffee
Traceback (most recent call last):
  File "/home/erik/mu_code/menu_function.py", line 18, in <module>
    drink = menu("Erik's drinks", drinks, "Choose your drink: ")
  File "/home/erik/mu_code/menu_function.py", line 9, in menu
    answer = choices[int(choice) - 1]
ValueError: invalid literal for int() with base 10: 'coffee'
>>>
```

"What are you doing???" Erik was enraged. "You should enter only numbers and not words!"

"But you gave me a list and asked what I want. I wanted coffee so I entered 'coffee.' What's wrong?" Simon tried to look innocent but he couldn't hide his smile.

"Well, for such *stupid* users like you I will print in ALL CAPS that you should enter A NUMBER!" Erik grumbled.

"Okay, okay, let me try again," asked Simon. He started the script again and at the first menu entered: "42".

```
Choose your drink: 42
Traceback (most recent call last):
  File "/home/erik/mu_code/menu_function.py", line 18, in <module>
    drink = menu("Erik's drinks", drinks, "Choose your drink: ")
  File "/home/erik/mu_code/menu_function.py", line 9, in menu
    answer = choices[int(choice) - 1]
IndexError: list index out of range
>>>
```

"Again?? You broke it again??" Erik was ready to slap Simon's hands on the keyboard. "Didn't you see that there are only three choices? Why did you enter 42??"

"First, because 42 is my favorite number. Second, yes, it was my mistake. Users make mistakes, you know. Seriously, I wanted to show you that your program should be ready for that. You can print whatever you want, in all caps, but there *will* be users who won't read it. There will be users who make mistakes."

"What should I do about it?" Erik was still angry at his brother, but he tended to agree with him. He made mistakes with programs himself.

"You should check what the user enters and tell them if the input was wrong. Let's think what we can do here."

4.1 If your user doesn't do what you expected

"What do you think the user should enter in the first menu?" Simon continued.

"They should enter 1, 2, or 3," Erik answered.

"Okay, so we can check if their answer was '1', or '2', or '3', then we pass it and pick that item from the list of options. But is it's not, we should tell the user that something is wrong."

"Yes, I remember, we can use `if-else` in Python," Erik suggested.

"Okay, let's try it," Simon said. "How are you going to do that? Try to explain it to me as if I didn't know about `if-else`."

"I will add to my function: 'if the user's choice is 1, or 2, or 3, then go ahead and use it. If not

(else), print that the user should enter one of those numbers'."

"Good, let's code it," Simon said.

Erik opened his editor and changed the function (four lines just before `return`):

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)

    if choice == '1' or choice == '2' or choice == '3':
        answer = choices[int(choice) - 1]
    else:
        print("Enter number 1, 2, or 3!")

    return answer
```

"Now let's try it," Simon said.

Erik started the program and at the first menu he typed: "2".

"Why did you enter '2'?" Simon asked.

"Because I wanted coffee," Erik answered.

"But we should test your program for *wrong* answers!" Simon said. "Yes, I know, it's your program and you don't want to break it, but as a developer you *have* to try to break your program. You have to imagine all possible ways your users will use your program in a wrong way. It's hard and very uncomfortable, I know, but you have to overcome it and try to enter all possible wrong values."

"Okay, okay," Erik said and restarted his program. At the first menu he entered "42" like Simon did the last time.

```
Erik's Menu
-----
1 chocolate
2 coffee
3 decaf
Choose your item: 42
Enter number 1, 2, or 3!
Traceback (most recent call last):
  File "/home/erik/mu_code/menu_function.py", line 21, in <module>
    drink = menu(drinks)
  File "/home/erik/mu_code/menu_function.py", line 14, in menu
    return answer
UnboundLocalError: local variable 'answer' referenced before assignment
>>>
```

"Let's see what's going on here," Simon said. "First of all, when you entered '42' your program

printed the message that the user should enter only 1, 2, 3. This is good. But then something went wrong. Look, it says that the variable `answer` was referenced before assignment:

```
UnboundLocalError: local variable 'answer' referenced before assignment
```

In simple words it means that you didn't create `answer` but you tried to use it. And Python shows you exactly where: you tried to `return answer` but Python didn't know anything about the variable `answer`."

"But why?" Erik said. "I have this `answer =` line in my code."

"Yes, you have it, but the important thing is *where* this line is used. In your code you create the `answer` variable *only* when the user entered the right choice. If the user entered something else, `answer` is not even created."

"In other words," Simon continued, "even when the user answered with wrong number, or even a word, you still had to return *some* answer. It's a very important rule: never use a variable before you create it and assign *some* value to it. What value can we assign to `answer` here in case the user made a mistake? I think an empty string like `''` should work here. Add it to your function and check if it helps."

Erik changed the function to this:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    choice = input(prompt)

    if choice == '1' or choice == '2' or choice == '3':
        answer = choices[int(choice) - 1]
    else:
        print("Enter number 1, 2, or 3!")
        answer = ''

    return answer
```

YOUR TURN Add checking the user's answer

In your `menu` function, add the code that Erik just added. Test whether it really checks your answers.

He tested the program again and this time it didn't give him an error. It printed the message `Enter number 1, 2, or 3!` and jumped to the next menu.

"Do you think your program did it right?" Simon asked.

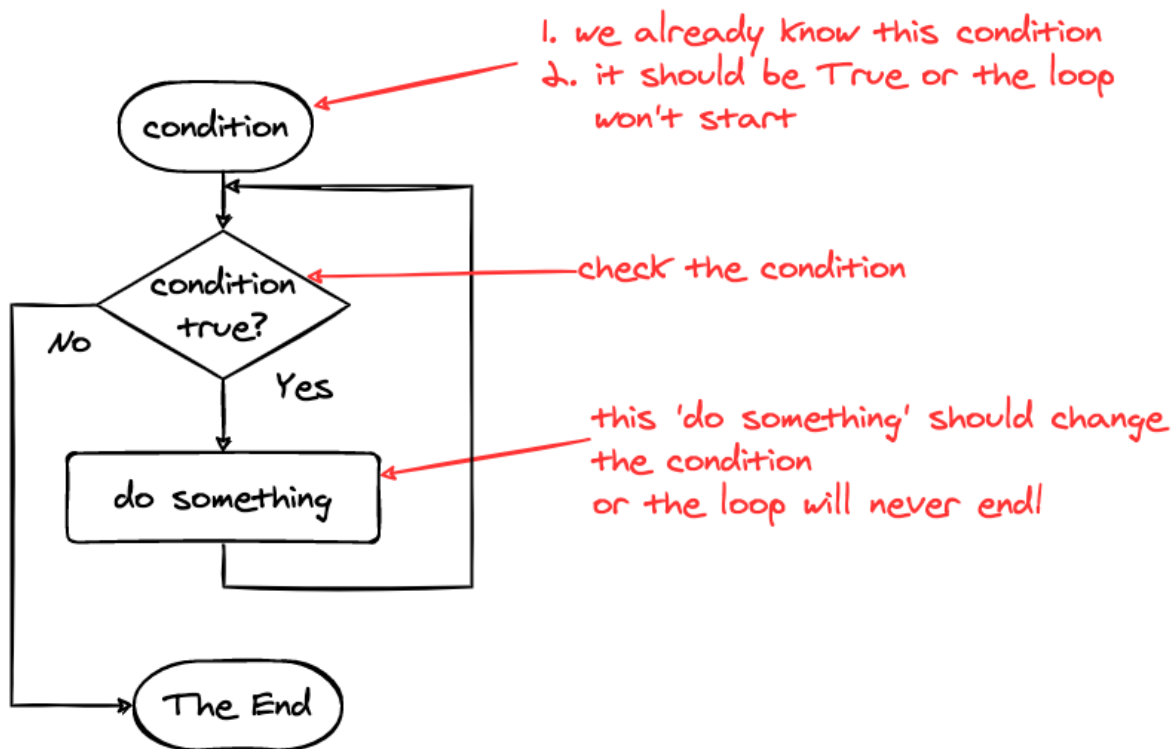
"It printed the message that you should enter 1, 2, or 3," Erik answered. "I think it's right."

"But you didn't get the user's choice for drink. If they entered a wrong number, you should give them a chance to enter a right one. It's not a test like in school where you have just one chance to answer. You should keep asking the user until you get one of the right answers."

"How should I do that?" Erik asked.

"We have another kind of *loop* for that," Simon started to explain to his brother. "It's called a *while loop*. It repeats something over and over again, and with every cycle it checks the *condition*. When the condition is true, it continues. If it becomes false, the while loop ends.

Sometimes we check the condition in the beginning on the loop if we already know it. In this case we say 'while something is true, do this'. But sometimes, like in our case, we don't have the answer when we start the loop because we haven't asked our user about their choice yet. So we start an *infinite loop* and check the condition *inside* the loop after we receive the answer from the user. We exit the loop if the condition becomes true. We call it *break* from the loop. Let me show it on a diagram," Simon said and started drawing.



EXAMPLE

$n = 5$

```
while n > 0:
    n = n - 1
```

Simon explained, "In this example we already know the condition before we start the loop. We check the condition and decide if we should start. It should be `True`, otherwise the loop won't even start. I used a *rhombus* figure for this check; this is how programmers usually draw decision points.

"If it's true then we do something. The important thing is that this 'do something' should change the condition, among other things. Otherwise the loop will continue forever and we don't want that.

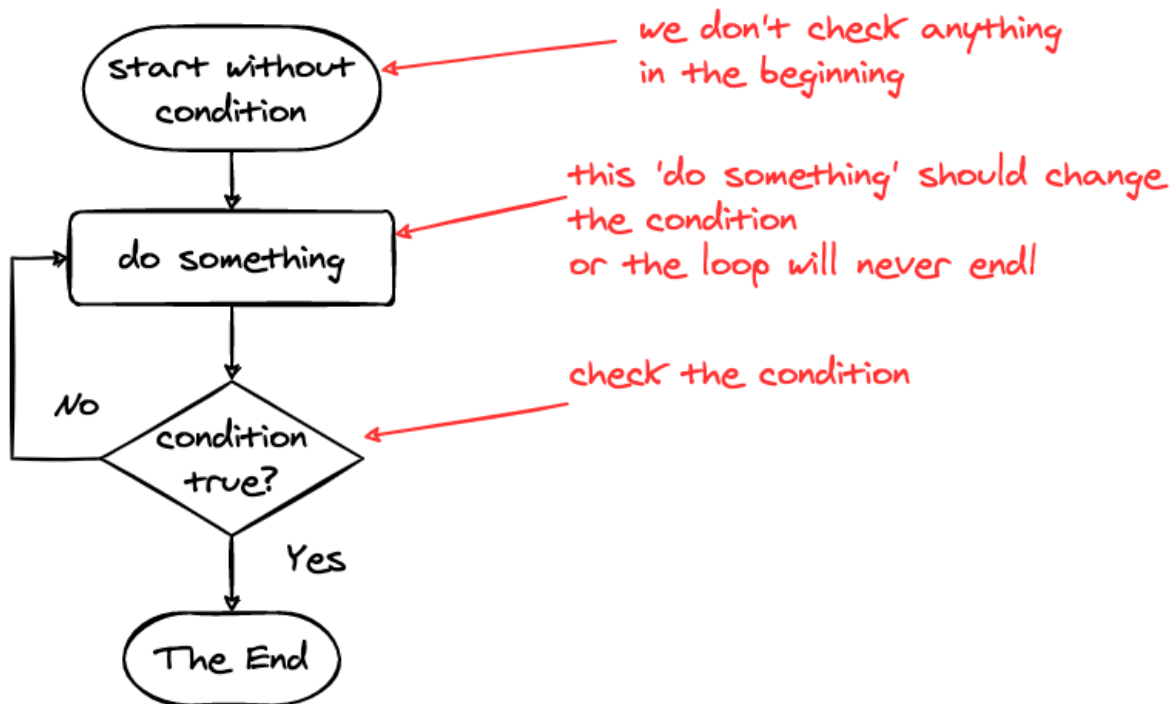
"After we have done that 'something,' we go to the beginning of the loop and *check the condition again*. If it's still true, we repeat that 'do something.' If not, it's the end of the loop and we go outside it to continue with our program.

"Look at this simple example: we want to count down from five to zero. First we set the variable `n` to five and in the beginning of the loop we check if it's greater than zero. Yes, it's greater, so we start the loop. Remember that we have to change the condition at some point, otherwise the

loop will never end. In our example, we subtract 1 from n each time we go through the loop. So, eventually, the variable n will become equal to zero and the loop will stop.

"The important thing is that in this case, before we start the loop, we already know what is in the variable n and we know that it's greater than zero.

"What if we don't know what's in the variable or the variable doesn't even exist? Like in our case: we can check the user's answer only *after* we ask to choose one of the items from the menu. So we have to do this," and Simon drew another diagram.



EXAMPLE

```

while True:
    answer = input()
    if answer == 'I':
        break
  
```

Simon continued his explanation, "Here we start the loop without checking any condition. We do something first and only *after that*, we check the condition. Sometimes this loop is called 'do-until,' which means 'do something until the condition is true.' When the condition is true, exit the loop and continue with the rest of the program.

In the example here I used your situation with `input()`. You ask for input, then you check that input. This is your condition: if the input is valid, you should exit the loop. In Python we use the

operator `break` for that."

Erik was a bit overwhelmed by this long explanation, but he felt like he knew what to do. He asked his brother, "So you're saying that I should just put a `while` in front of my `input()` line and add a `break` after I get a correct answer?"

"Yes," Simon answered, "exactly right! Just don't forget to indent all the lines that are part of the loop by four spaces to the right. Your editor will help you, don't worry."

Erik started working on his program. After several minutes his function looked like this:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        if choice == '1' or choice == '2' or choice == '3':
            answer = choices[int(choice) - 1]
            break
        else:
            print("Enter number 1, 2, or 3!")
            answer = ''

    return answer
```

"Right?" Erik asked his brother.

"I told you that your editor can help you. Click Check  Check ."

Erik did as his brother suggested and saw this:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i += 1
    while True:
        choice = input(prompt)
        if choice == '1' or choice == '2' or choice == '3':
            answer = choices[int(choice) - 1]
            break
        else:
            print("Enter number 1, 2, or 3!")
            answer = ''

    return answer
```

↑ Syntax error. Python cannot understand this line. Check for missing characters!

↑ Expected an indented block


"You see now?" Simon said. "You forgot to indent the lines to the right. That's why it says that it expects an indented block here. Move all the lines that are part of the loop to the right."

Erik changed his function and checked the code again.

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        if choice == '1' or choice == '2' or choice == '3':
            answer = choices[int(choice) - 1]
            break
        else:
            print("Enter number 1, 2, or 3!")
            answer = ''
    return answer
```

YOUR TURN Add the while-loop to your function

Add the loop like Erik just did. Try to use the Check button to find if there are errors in your code. Try to make errors and see if the editor can find them.

Now the Check button showed a green thumbs-up  Check good for a moment so Erik knew it was okay. He clicked Run to test the program. At the first menu he entered '42' like Simon did last time. The program reported the he should choose a number 1, 2, or 3 and returned to the prompt again! It didn't crash, it worked!

"What was the other wrong thing Simon did with my program?" Erik tried to remember. "Right, he tried to enter 'coffee'! Let's try that too."

He entered 'coffee' and his program responded as he expected again! Finally Erik typed '2' and got the next menu. Here is what he saw in the editor window (we marked Erik's input with **bold**):

```
Erik's Menu
-----
1 chocolate
2 coffee
3 decaf
Choose your item: 42
Enter number 1, 2, or 3!
Choose your item: coffee
Enter number 1, 2, or 3!
Choose your item: 2
Erik's flavors
-----
```

At the next menu he typed 4 for Raspberry and got the same message:

```
Erik's flavors
-----
1 caramel
2 vanilla
3 peppermint
4 raspberry
5 plain
Choose your flavor: 4
Enter number 1, 2, or 3!
Choose your flavor:
```

"Why is that?" he asked Simon.

"Your program behaves exactly as you wrote it," Simon answered. Of course, he knew where the problem was.

"How did you write your condition?" he asked.

Erik said, "If the answer is 1, 2, or 3. A-ha, I understand now! I entered '4' so the program thinks it's a wrong answer! But how can I fix this?"

"It looks like we need a list of valid answers for each menu list," Simon said. "You can pass it as another argument. But I think you know enough already to create a better solution."

"What is it?" Erik asked. "Something with loops again?"

"Not only," Simon answered. "We'll learn something new about lists, too. Yes, I see that you are tired of my lectures, but let's finish it today. It will make your program work right again—isn't it worth it?" and he winked at his brother.

Erik was tired indeed, but that bug (yes, he has learned that word!) with the flavors menu was really annoying and he wanted to fix it. "Okay," he sighed, "let's fix it. What did you want to tell me about lists?"

"Look at your condition," Simon said. "You used a simple 'if-else' check and you checked the input against three valid answers: 1, 2, and 3. But what if your list of items is long, like 20 items? Your 'if-else' block will be too long. There is another way in Python. We can check if a certain item is in the list. In our case, we can check if the answer we got from the user is in the list of 1, 2, 3. We can try that for the first menu. Let me show you."

Simon took Erik's keyboard and changed the `if choice == ...` line in his function to the following:

```

. . .
while True:
    choice = input(prompt)
    if choice not in ['1', '2', '3']:
        answer = choices[int(choice) - 1]
        break
    else:
        print("Enter number 1, 2, or 3!")
        answer = ''
. . .

```

"This needs some explanation," he said. "Look, the user enters a string that can be '1' or can be '42'. We test whether this string is in the list of allowed answers, which is 1, 2, 3. If it's not in the list, then we print the error message and continue with the loop. If it *is* in the list of allowed answers, then we convert it and pick that item from the menu list."

"Yes, I understand," Erik said. "But what about the second menu where I got the error? It didn't let me choose number 4 because it wasn't in my `if`. With your list I will have the same problem. I have five flavors, so I need another list of answers here, right?"

"Excellent question!" Simon said. "I was just about to ask it myself. Yes, you are right. Each menu list should have its own list of allowed answers. Not a big problem; we can build one when we know what's in our menu list. Let me write it first and then I'll explain it step-by-step." And Simon added the following lines just before the `if` block he added previously:

```

. . .
while True:
    choice = input(prompt)
    allowed_answers = []
    for a in range(1, len(choices)+1):
        allowed_answers.append(str(a))
    if choice not in allowed_answers:
        answer = choices[int(choice) - 1]
        break
    else:
        print("Enter number 1, 2, or 3!")
        answer = ''
. . .

```

"First (look at the label **1**) we create an empty list for allowed answers. Then (label **2**) we measure the length of the menu list using the function `len()`. For your drinks, the result will be three, and for the flavors it will be five. Then we use the function `range()` to create a *sequence* of numbers from one to the length of the menu. For drinks the sequence will be 1, 2, 3. For flavors it will be 1, 2, 3, 4, 5. You get the idea. Just notice that in the `range()` function we shouldn't use the *last* element of the sequence, but the one *after the last* which is not included in the sequence. That's why we have to add one to the length of the menu like this:"

```
len(choices)+1
```

"And finally in this loop (label **3**), we convert each number from the sequence to a string and add it to the end of the list of allowed answers. This function is called `append()`." Simon finished his

explanation and added: "Yes, it's a bit complicated for the first time, but try to read the Python code here yourself and you'll understand it as if it's plain English."

"Now we have to change our list with 1, 2, 3 to the list of allowed answers that we just built," Simon made that change, looked at the code, and slapped his forehead: "Oh, I just noticed!"

"What?" Erik thought they have finished already. But it looked like there was something else.

"We also have to change our message," Simon said. "Because our function now can accept menu lists of any length, we should tell the user something like: 'Enter a number from 1 to 6' or 'from 1 to 12,' depending on the length of our menu. Remember how we can get the length of a list?"

"With the `len()` function?" asked Erik.

"Of course!" Simon said, and made the final change in the function. It now looked like this:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        allowed_answers = []
        for a in range(1, len(choices)+1):
            allowed_answers.append(str(a))

        if choice in allowed_answers:
            answer = choices[int(choice) - 1]
            break
        else:
            print("Enter number from 1 to ", len(choices))
            answer = ''

    return answer
```

YOUR TURN Add the `allowed_answers` list

Add the list of allowed answers to your function. Test whether it allows you to use menu lists of different length.

"Now we are protected from user errors!" Simon said. "Try it and see if it works!"

Erik ran the program again and entered '42', 'coffee', 'weryiuryt587456' but the program didn't crash like before. Every time, it gave him a reminder that he should use a number and it should be between 1 and 3 or 5, depending on the menu.

"This is cool! It works and it doesn't crash!" Erik was really happy that he created such a robust program.

"One more thing," Simon said. "This one will be really-really final for today, I promise!"

"Okay," said Erik. He started to like this programming thing. He liked that his program now looked like a real one, and it worked! Even if it wasn't an online or mobile application yet, it worked like a chat with a shop. Erik imagined that he was texting with his favorite coffee shop, ordering his drinks, and then coming to pick them up.

Simon said, "Your program now doesn't let me enter anything except the numbers from 1 to 3 or 5. But what if I want to skip something? Like I don't want any toppings on my drink?"

"In my flavors menu I have 'plain,' which means 'no flavor.' I can add the same to toppings," Erik answered.

"That works too," Simon said. "But in general with every menu you should give your user an option to exit the menu. Usually people use something like 'Click X to exit from this menu.' I think we should add this to our menu function too."

"How do we do it?" Erik asked. He thought a little and said, "I know! We will add 'X' to the list of allowed answers! Am I right?"

"Absolutely!" Simon was very glad to see his brother making such a good progress. "Remember, we used the function called `append()` to add items to the list? We can use it here, right after we finished adding numbers to the `allowed_answers`."

"Let me try," Erik said and started typing.

"Sure, go ahead," Simon encouraged his brother. "Just make sure you add it *after* that for-loop. Better to add an empty line after it; that way you will be able to see that it's not part of the loop."

Here is Erik's new version of the menu function:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        allowed_answers = []
        for a in range(1, len(choices)+1):
            allowed_answers.append(str(a))

        allowed_answers.append('X')

        if choice in allowed_answers:
            answer = choices[int(choice) - 1]
            break
        else:
            print("Enter number from 1 to ", len(choices))
            answer = ''

    return answer
```

"Nice," Simon said. "I would also add the lower-case letter 'x', because that's what most people would type. Now what should we do if the user types 'x'?"

"Exit the menu loop?" Erik said.

"Right! But what are we going to return to the main program? Normally we return the user's choice from the menu: coffee or chocolate or whatever. What if the user types 'x'? What should we return?"

"Nothing?" Erik suggested.

"Yes, we just return an empty string," Simon said. "If the user types 'x' you just assign to `answer` an empty string like this: " and return the answer the same way you return it if the answer is in the menu."

Simon continued, "Important thing: you should do that check *before* you try to convert it into a number, but *after* you check whether it's in the `allowed_answers` list. Do you see where to put this check?"

"Yes, right after this line: `if choice in allowed_answers.`"

"Great! Go ahead and add it! In this case you will have a *nested* if statement: one `if` inside of another `if`. This is pretty common, sometimes you see three levels of nested 'ifs' or even more. Just make sure your indentations are correct. This is how Python tells the computer what to do if the condition is true or false."

Erik worked more on his code and finally got this:

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        allowed_answers = []
        for a in range(1, len(choices)+1):
            allowed_answers.append(str(a))

        allowed_answers.append('X')
        allowed_answers.append('x')

        if choice in allowed_answers:
            if choice == 'X' or choice == 'x':
                answer = ''
                break
            else:
                answer = choices[int(choice) - 1]
                break
        else:
            print("Enter number from 1 to ", len(choices))
            answer = ''

    return answer
```

YOUR TURN Add the exit option

Add the 'X' option to the list of allowed answers. Add the nested 'if' to check it. Test whether it works. What if you enter 'x' in all menus? What order will you get in that case?

He tested the program, entering 'x' in all three menus and he got what he expected:

```
Erik's Menu
-----
1 chocolate
2 coffee
3 decaf
Choose your item: x
Erik's flavors
-----
1 caramel
2 vanilla
3 peppermint
4 raspberry
5 plain
Choose your flavor: x
Erik's toppings
-----
1 chocolate
2 cinnamon
3 caramel
Choose your topping: x
Here is your order:
Main product:
Flavor:
Topping:
Thanks for your order!
```

"An empty order!" he said.

"Right, exactly as it should be," Simon confirmed. "I like your program," he continued. "It works—that's the first and the most important part. It's user-friendly, and it gives the user instructions on what to do—that's the second part. It checks the input and it doesn't let the user enter wrong values—that's the third part."

"Let's quickly recap what you learned today," Simon said. "What was the first thing?"

"First, you broke my program again!" Erik answered. He wasn't very angry this time because he knew that together with Simon they have fixed the program. "And you told me that I should always think how my users can use program in a bad way."

"Right, dealing with stubborn users who don't want to follow your instructions is a part of programmer's job," Simon smiled.

"Then we wrote a menu loop where we checked what the user entered and didn't allow them to use answers that are not allowed. Then you told me about that `append()` function to add something to a list."

Erik continued, "Then I wrote *nested* if checks and now my code looks like real programs they show in movies."

Simon smiled, "Trust me, what they show in movies are very rarely real programs. But you are right, your program is getting more complex, it uses different Python operators, all these loops, and ifs, and lists."

"And then we added the 'x' option in the menu and now any user can get an empty order!" Erik giggled.

"Yes, why not," Simon said. "You shouldn't *force* your users to always order something. You should give them an option to cancel their order or to exit from the menu."

Simon continued, "The program is really good now. Tomorrow I'll ask you stop being a programmer and become the coffee shop manager."

"Am I not a manager already?" Erik asked.

"Yes, you are," Simon smiled. "Now imagine you, the coffee shop manager, just received a new flavor component for your coffee drinks. And you want to add that flavor to the menu. Oh, and also a couple of new toppings. What would you do?"

"I would add those toppings to the list of toppings, not a big deal," Erik answered.

"Yes, but what if you are *just* a manager and not a programmer? You know nothing about this

program, you don't know Python, but you want to add those flavors and toppings to the menu. You, as a programmer, should give the manager an easy way to add something to the menu."

"How do you suggest to do that?" Erik asked. He knew already that Simon has something in mind.

"I think we should put menus in files and read your lists from those files."

"Like in Word documents?" Erik asked.

"Yes, almost," Simon said. "Your program will open those files and read from them. I think the easiest will be to have one file per menu. One file will have all drinks, another one—all flavors, and another—all toppings. Then your manager will just edit those files instead of editing your Python code. Sounds good?"

"Yes, interesting," Erik said. He wondered how his Python program would open files the same way Word does.

"Great," Simon said. "This is what we are going to do tomorrow. Take some rest now."

4.2 New things you have learned today

- *Users make mistakes*
You have learned that user not always follow the directions you give them in your program. You have to be ready for that and check their input for errors, wrong types, etc.
- *Indentation*
When you create a *block* in Python (like `while`, for example) you have to make sure all code in the block is *indented*, i.e. shifted to the right.
- *How to exit from a menu*
You have to give your users a way to exit from each menu. For example, if they don't want to order any topping and they want to skip it.

4.3 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch04>

Working with files: Being a shop manager

This chapter covers

- Erik learns what a coffee shop manager needs
- Erik reads his menus from files
- Erik writes his second Python function

"Last time you said something about a coffee shop manager," Erik started with his brother the next day. "Something about changing menus. I forgot."

"Yes, I said it might be good if your coffee shop manager could change the menus without going into Python code," Simon said.

"Yes, good idea," Erik said. "Not everyone knows Python."

"My idea is to create simple text files for each menu—call them `drinks.txt`, `flavors.txt` and `toppings.txt`. Then your program can read from those files and create lists from the items in the files."

"Why did you name them all with the `.txt` at the end?" Erik asked. "Shouldn't they be `.docx` so the manager could edit them in Microsoft Word?"

"Good point," Simon said. "Yes, the manager might be more familiar with Word, but in our case we need just a *plain text file*, without fonts, or headers, or table of contents. It's similar to your Python code—these files should have nothing but lines of plain text, and the manager should use a plain text editor to work with them. When I name them with `.txt` at the end I tell the operating system—whether it's Windows, or macOS, or Linux-- that this file should be opened with a plain text editor and not a word processor like Word. In all those systems there is always a text

editor that can edit these files. And you can also install another application for that, like we with did with Mu Editor. Also, for Python it's much easier to read from a plain text file than from a .docx file."

"Let's create these files," Simon continued. "You can use your Mu Editor for that. Just don't forget to add the .txt extension when saving the files. Otherwise it will automatically add .py. Create a new file, enter your drinks, each on a separate line, and save it with the name drinks.txt. Then do the same for flavors and toppings."

Erik started working. After several minutes he's got three files.

Listing 5.1 drinks.txt

```
coffee
chocolate
decaf
```

Listing 5.2 flavors.txt

```
caramel
vanilla
peppermint
raspberry
plain
```

Listing 5.3 toppings.txt

```
chocolate
cinnamon
caramel
```

YOUR TURN Create your own menu files

Create the text files with menu items like Erik just did. Make sure they are plain text files. Try to use your favorite flavors and toppings in your menus.

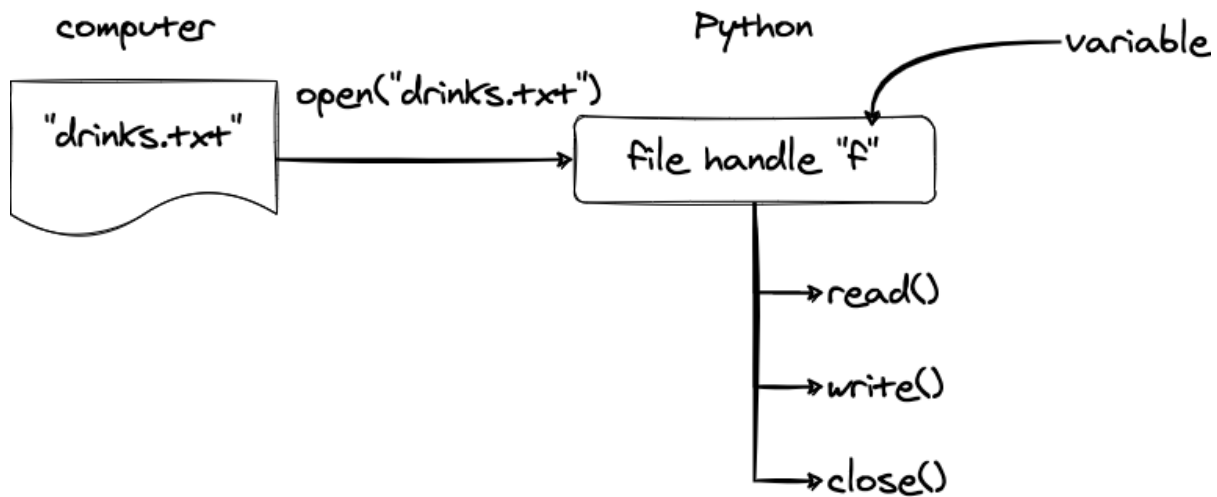
"Now what?" he asked his brother.

"Now let's learn how to work with files in Python. Let's create a new program for that. We'll practice a little bit and then we'll add this code to our main program. I usually do this when I learn something new—try it in a separate simple program before adding to the main application."

Simon continued, "Create another file in the editor and save it as files.py."

Erik has done that several times already so it took him just a couple seconds.

"Working with files is a difficult topic, so let me start with a diagram," Simon said.



"When you work with a file in your computer you use its name. You tell your editor program to *open* a file called 'drinks.txt'. Your editor program then *reads* the file and shows you its content. Then you edit the file and save it which means you *write* the file on the computer disk. So far, so good?" Simon asked.

"Yes," Erik said. "But disks are used only in very old computers. In my computer it's called SSD drive and it's not a disk anymore. My friend Alex told me."

"You're absolutely right!" Simon was glad to hear that from his brother. "Yes, it's SSD now in most of the computers and yes, it's not a disk and it is not spinning. By the way, maybe your friend Alex would like to join us? It seems that he is interested in computers. Maybe he wants to learn some programming too?"

"I'll ask him," Erik said. "But let's get back to my files. I see that you wrote 'computer' and 'Python' at the top of your picture. What does that mean?"

"That means," Simon started his explanation, "that in Python if you want to work with a file you have to create a special *object* which is usually called a *file handle*. You use this object to read and write the file. You use a function called `open()` to create such an object. You call the function `open()` and you pass the file name as an argument. In our case it will be `open("drinks.txt")`. The function returns the file handle which you put in a variable. In this case the variable is called `f`, but you can use any name here."

"Why is it so complicated?" Erik asked. "Why can't we just use the file name?"

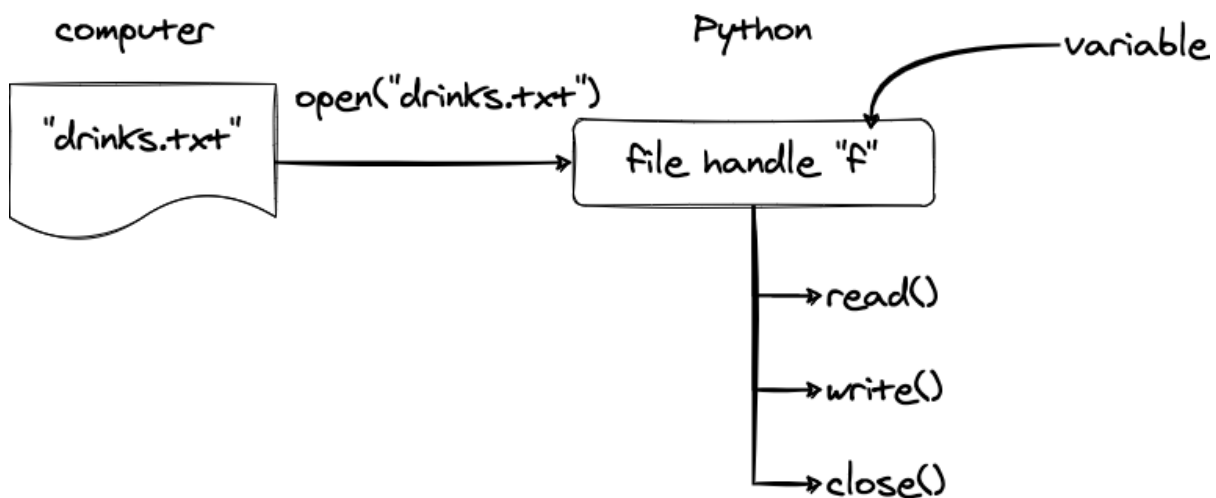
"Yes, it looks a bit complicated for the first time. The reason for that is that the file name is just a string, remember? When we want to read from the file we want to read from *the file with that name*, not from the string. The file *name* and the file *itself* are different things. When we use the

function `open()` we create a connection between the file name and the file itself. We tell Python: 'please find the file named `drinks.txt` inside the computer and use it *as a file*.' Don't worry too much about it right now. Sometimes the best way to understand something is to start using it."

"Okay," Erik said. He was still a bit confused about all that but we wanted to see how he can read his menus from the text files he just created.

"Now let's write a simple Python program to work with files," Simon said. "Now go to the tab in your editor where you have your `files.py` program opened.

Look at my diagram: you have to call the function `open()`, pass the file name such as '`drinks.txt`', and store the result in the variable `f`. Can you write it?"



"Let me try," Erik said and wrote this.

```
f = open("drinks.txt")
```

"Good!" Simon said. "Now you have a file object named `f` and you can read from it. To do that you call a *method* called `read()`. Methods look and behave very similar to functions, but methods are applied to objects. We'll talk about objects later but for now all you have to know is that to call a method of an object you use the object's name, then you put a dot and then the method name with parentheses, similar to calling a function. Like this: `f.read()`. Methods also can return something, similar to functions. So you have to store the result somewhere. What do you think this method `read()` will return when you call it?"

"What's inside the file, I think," Erik answered.

"Absolutely correct!" Simon said. "Save it in a variable called `drinks` and then try to print it."

Erik wrote the following.

```
f = open("drinks.txt")
drinks = f.read()
print(drinks)
```

"Now try to run it," Simon suggested.

Erik clicked Run and got the output.

```
coffee
chocolate
decaf
>>>
```

YOUR TURN Read from file

Write the same short program and try to read from the 'drinks.txt' file. Make sure you can print out the whole file's content.

"It works!" he was really glad. His Python program opened a file, read it, and printed it on the page! "Now I know how to print my menus from Python! Let me write the same for the other two files!"

"Right," Simon said. "But that's not exactly what we want."

"Why?" Erik couldn't understand.

"Remember, in your program you don't only print out your menus, but you also let the user choose from the menu and then you find that item in the list, right?"

"Yes, but isn't it a list here? It looks like a list," Erik asked.

"It may look like a list, but it's not a list. It's a string," Simon said. "When you called the `read()` method you copied the whole file's content into a variable called `drinks`. So this variable is just one large string. If you don't believe me, you can test it right here. See these three angle brackets in the output window? You can type any Python command here to continue working with your program, the same way we did it with REPL, remember? Type here this: `type(drinks)` and you will see the type of this variable."

Erik did and saw this:

```
>>> type(drinks)
<class 'str'>
>>>
```

"You see, Python says it's a string," Simon said. "And we need a list."

"What should we do?" Erik asked.

"Luckily, Python developers knew that we might need this and created another method for the file object. It's called `readlines()`. Try to change your `read()` to `readlines()` and see what happens. Don't forget to click Stop before running your program again."

Erik changed his program to the following.

```
f = open("drinks.txt")
drinks = f.readlines()
print(drinks)
```

He clicked Stop and then Run again and got this output.

```
['coffee\n', 'chocolate\n', 'decaf\n']
>>>
```

"Try to check its type again," Simon suggested.

Erik switched to the output window and typed:

```
>>> type(drinks)
<class 'list'>
>>>
```

YOUR TURN Check Python types

Repeat the checks that Erik just did. Do you see the difference between a string and a list?

"It's a list!" he said. "But what are those *slash-n* characters? I don't have them in my file with drinks."

"You don't *see* them in your file, but they are there. These are *invisible* characters. When you see this *backslash-n* it's a single character that is called a 'newline' character. It tells the computer that it should print the next item at the beginning of the next line. Without it all your drinks would be printed like this: `coffeechocolatedecaf`. You don't want that, right?" Simon smiled.

"Of course, not!" Erik said. "But we don't need them in the list, right? I think the menu lists should look like in my main program, right?"

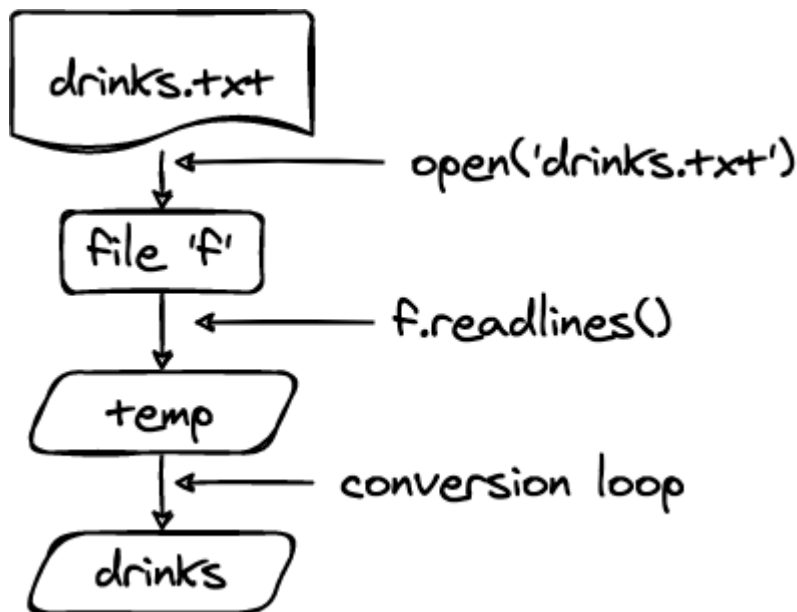
"You are absolutely right. And again, Python developers created a very useful method for that. It's called `strip()` and it removes invisible characters from both ends of the string. We just have to apply it to every item in the list. What do you think we should use here?"

"A loop?" Erik suggested.

"Right, a loop!" Simon said. "We will go over the list and remove those newline characters with the `strip()` method."

Simon paused for a moment, thinking. Then he continued, "There are several ways to do it. Some are shorter, but they are more difficult to understand. Let's use the one that is easier to read and follow. Actually this is a good rule in programming: when choosing between different ways of doing something, always use the one that is easier to read and follow. If somebody is reading your code, they will thank you for that. Even yourself—if you are reading your own code three months later."

He quickly drew a diagram.



"Let's use a temporary list to read into from the file. Then we'll go through that temporary list, convert each item, and append it to the new list. And that new list we'll call `drinks`. Then we'll repeat the same for flavors and toppings. We can use the same temporary variable for all of them. Let me help you," and Simon started typing in Erik's program. Here is what it looked like after he finished.

```
f = open("drinks.txt")
temp = f.readlines()
drinks = []
for item in temp:
    new_item = item.strip()
    drinks.append(new_item)

print(drinks)
```

He clicked Run and they saw the result:

```
['coffee', 'chocolate', 'decaf']
>>>
```

YOUR TURN Remove the newline characters
Remove the newline characters from the menu items using the `strip()` method.

"Looks better, doesn't it?" he asked Erik. "Now go ahead and do the same thing for the other menu files."

Erik started working on his program and when he was almost done with the `flavors.txt` file he exclaimed, "Wait! I am repeating myself! You told me I should not repeat myself."

Simon smiled. He was happy that his little brother grasped this concept so quickly. "What should we do to *not* repeat ourselves?" he asked.

"Write a function?" Erik said.

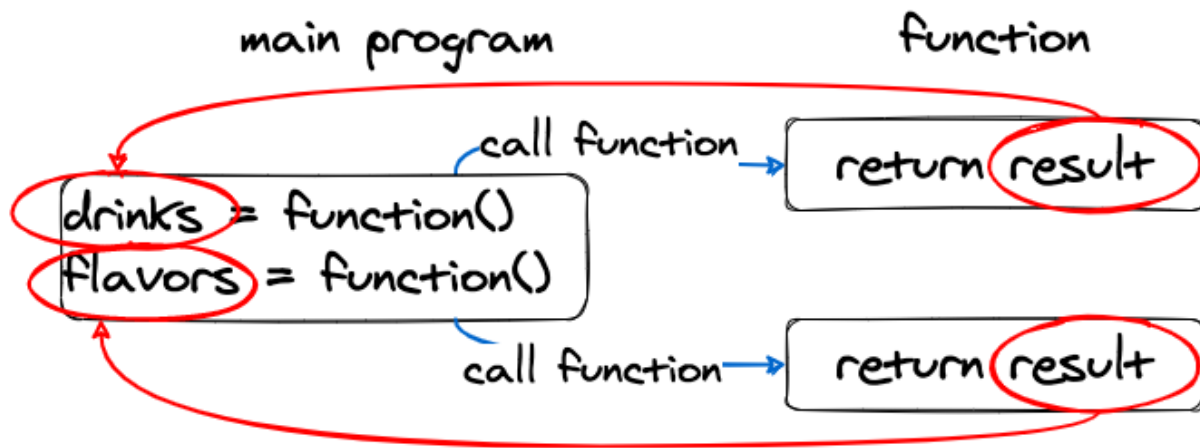
"Yes, exactly! Look at this code: what is the same and what is changing? What is going to be an argument and what is this function going to return?"

Erik started thinking out loud, "I open different files, so the file name should be an argument, right?"

"Correct!" Simon confirmed. "What do you want to return?"

"I think I return the list with menu options. After we removed those newline characters, of course. But how should I call this list?"

"You can call it any way you want because it's not *visible* from the outside. Call it `result`, for example. Then you can write `return result` at the end of the function. When you call the function that `result` list is *assigned* to, the variable in the main program, like `drinks` or `flavors` depending on the file you are reading from. *Inside* the function the variable will be always called `result`, but *outside* the function, in the main program, you can assign the result to any variable."



"Go ahead and write that function!" Simon encouraged his brother. "Remember how to do it? Start with `def`; name the function `read_menu`, for example; pass `filename` as an argument; and copy the code we just have written."

"Okay, I'll try," Erik said and started writing his second Python function.

Simon helped his brother a little bit and here is what they wrote together.

Listing 5.4 `files.py`

```
def read_menu(filename):
    f = open(filename)
    temp = f.readlines()
    result = []
    for item in temp:
        new_item = item.strip()
        result.append(new_item)

    return result

drinks = read_menu("drinks.txt")
print(drinks)
flavors = read_menu("flavors.txt")
print(flavors)
toppings = read_menu("toppings.txt")
print(toppings)
```

Erik saved the file and clicked Run. Of course, he got the expected result.

```
['coffee', 'chocolate', 'decaf']
['caramel', 'vanilla', 'peppermint', 'raspberry', 'plain']
['chocolate', 'cinnamon', 'caramel']
>>>
```

YOUR TURN Create the `read_menu` function

Create the `read_menu` function the same way Erik did. Make sure you don't have typos in the file names. What if you do? Try to change the file name and see what error Python gives you. Don't forget to fix the file name so your program works again.

"It's my second function and it works!" he proudly said to his brother.

"Yes, you are building your own function library already, great!" Simon said. "Now let's copy your new function to the main program. Don't copy the `print()` lines—we used them just for testing. I think that file is called `menu_function.py`, right?"

"Right," Erik said. "But where should I put my function in that file? In the beginning or in the end?"

"The rule in Python is that you should define your function *before* you start using it. Because of that usually all functions are placed in the beginning of the file, before the main program. You can place it right after your first function."

"Okay," Erik said and started working. Here is what he's got.

Listing 5.5 `menu_files.py`

```
def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    . . .
    # this function didn't change
    . . .
    return answer

def read_menu(filename):
    f = open(filename)
    temp = f.readlines()
    result = []
    for item in temp:
        new_item = item.strip()
        result.append(new_item)

    return result

drinks = read_menu("drinks.txt")
flavors = read_menu("flavors.txt")
toppings = read_menu("toppings.txt")

drink = menu(drinks)
flavor = menu(flavors, "Erik's flavors", "Choose your flavor: ")
topping = menu(toppings, "Erik's toppings", "Choose your topping: ")

print("Here is your order: ")
print("Main product: ", drink)
print("Flavor: ", flavor)
print("Topping: ", topping)
print("Thanks for your order!")
```

YOUR TURN Copy your function to your main program

Copy the new `read_menu()` function to your main program and try it.

He tested the program and it worked exactly as before!

"This is good," Simon said. "Now try to add something to the toppings file, for example. And see

if it changes the menu."

Erik opened the `toppings.txt` file and added 'vanilla powder' to the end of file and saved it. He ran the program again and indeed there was the additional line at the last menu: 4 vanilla powder!

YOUR TURN Add another item

Add a new item to one of the menus. Change one of the items. Don't forget to save the menu files after you changed them. Check whether your program prints the updated menus.

"This is good, I like it!" he said. "Now anybody who can edit a text file can change the menu! Wait..." he had an idea. "So I can put *anything* in these menus! Ice cream or sandwiches or... Cool, I like it! I should tell my friend Alex about it—he likes LEGO minifigures. Maybe he can use this program to exchange figures with friends!"

"Exactly right!" Simon said. "I'm glad you have so many ideas on how to use your program, this is great! I have some ideas too but we'd better talk about them tomorrow. Also tomorrow we have to create the main menu."

"What do you mean?"

"You see, currently you have to start your program every time you want to take an order. You take the order, you print it, and your program finishes. It would be better if your program could return to the initial dialogue where you ask the customer's name."

"Yes, right," Erik agreed. "It should be like a kiosk where you order something, press 'Done', and it goes to the first screen with 'Welcome to our shop'. Yes, let's do it!"

"Let's recap today's progress," Simon suggested. "What did we do today?"

"First, you said that the coffee shop manager will want to edit our menus in files. Then I wrote three files with menus for main drinks, flavors, and toppings."

"Very good, what's next?"

"Then I opened the files and read from them. I could read line by line but then I had those strange 'backslash-n' characters. And then we used the `strip()` method to remove them."

"Good," Simon said. "And did you remember what I told you about objects?"

"Not really. You said that the file is an object in Python and it's not the same as its name. And also you said that functions with objects are called 'methods'."

"Yes, everything is right," Simon said. "Objects is a difficult topic, we will talk about it more later. For now we just use them and their methods, but we will learn more about them later. And we'll create our own objects and methods too—like we did with functions."

"Right!" Erik said. "You reminded me—I wrote my second Python function! And it worked!"

"Indeed! You are becoming a serious programmer now!" Simon said and smiled. "Let's take a rest for now. Tomorrow we'll make your program even better!"

5.1 New things you have learned today

- *What it means to open a file*
You have learned the difference between the file name and the file handle inside the program.
- *What is `\n` and how to remove it from strings*
You have learned that the `\n` symbol means "start a new line". We don't need it in our menu items so we used the `strip()` function to remove it.
- *What you return from a function is assigned to a variable in the main program*
You have learned that the variable inside a function is not *visible* in the main program. To pass its value we have to *return* that variable from the function and *assign* its value to another variable in the main program.

5.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch05>

Main menu: Next customer!

This chapter covers

- Erik creates the main menu to serve many customers
- Erik learns about Python dictionaries
- Simon explains the 'top-down' development approach

"Remember, we decided that we have to create the main menu?" Simon asked Erik.

"Yes, you said if I want to use this program to serve many customers, I have to repeat the menus for each customer. Ask their name and what they want to order."

"Exactly right!" Simon said. "And what are you going to use for that?"

"A loop, maybe? Like we did with menus. Repeat until the customer types the right numbers or types 'X'."

"You are absolutely right!" Simon was really glad that his brother caught this programming idea so quickly. "We will ask the customer their name, like in our first program, remember? Then we'll get their order with all the flavors and toppings."

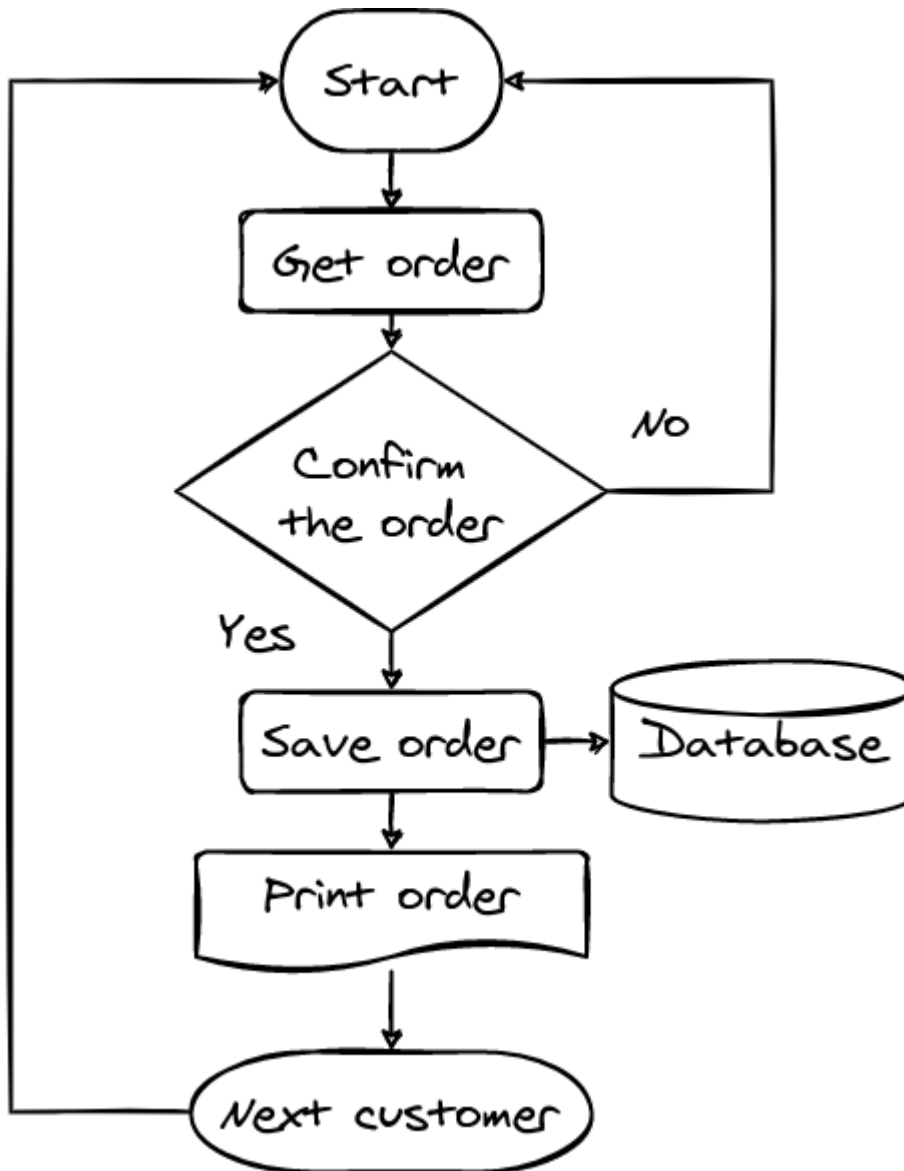
"Yes," Erik continued, "and then we'll ask them: Order or Cancel? I saw that on some web sites."

"Right. When they click Order, we save the order and print it for the barista. If they cancel, we just forget about it. In both cases we go back to the beginning of the main menu and ask the next customer their name."

Simon took a piece of paper and started drawing. "We should first plan this algorithm visually.

When we both agree how it should behave you can start writing the code. It's always a good practice to discuss your future program in plain words and diagrams before you start writing any code."

"The first thing we do here is getting an order," Simon started explaining. "You see, I put it here as 'Get order'."



"And where are all our menus with flavors and toppings? Why didn't you put them here?" Erik asked.

"I decided to use a block called 'Get order' that *contains* all the menus. It's a common way to think in big blocks first and then work on each block's details separately. By the way, it's

another reason why programmers use functions. They think about the program in big blocks first and then describe each block in a separate diagram. Imagine if we placed every minor detail of our program on a single diagram! It would be impossible to understand the main algorithm!"

"Let's continue," Simon said. "So we get the order and we ask the customer to confirm it. They can cancel and then we return to the very first menu: 'Welcome to Erik's Coffee Shop' and all that.""

"If the customer confirms the order," Simon continued, "then we should save and print it."

"Okay, I understand that we should print it to prepare the drinks," Erik said. "But why should we save it? And what do you mean by *saving* the order?"

"First of all, it would be good at the end of the day check how many of your friends you served in your coffee shop, don't you think?"

"But I know already," Erik said. "I prepared five drinks that day."

"But we're talking about a *real* coffee shop, don't forget. They work every day and they serve tens and hundreds of customers. A couple of my friends worked in different coffee shops and I can assure you—they know very well how many customers they serve each day."

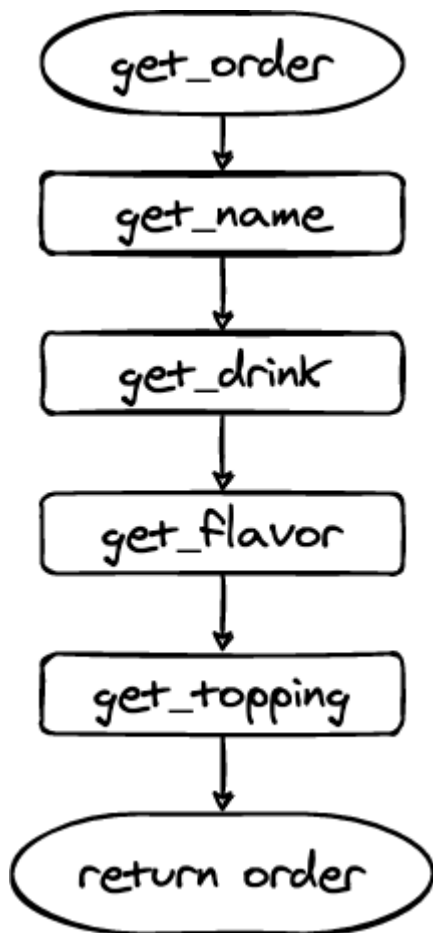
"The other reason," Simon continued, "is that the coffee shop manager should know what they have in the shop what they should order for their inventory. Remember we talked about changing menus? For example, they didn't order the caramel flavor in time and they had to remove it from the menu. Why didn't they order it? Because they didn't count how many portions of caramel flavor their customers have ordered. So we have to save all orders, analyze them, and order flavors and toppings if we don't have enough."

"I didn't think about it," Erik said. "Yes, it's a good idea to save the orders; you are right. But how are we going to do that?"

"There are several ways," Simon said. "We can use files, or we can use databases. Of course, all serious applications use databases. I think we should start with files and then, if you are brave enough, we can use databases too."

"Yes, I want my program be like those *serious* applications!" Erik said. "I want to try databases too!"

"Good," Simon said, "but for now let's finish with the main menu. We'll get to saving orders very soon. Speaking of orders," Simon took another piece of paper, "here is what's inside that 'Get order' block."



YOUR TURN Create your own diagram

If you decided to work on a different kind of shop, create a diagram for your `get_order()` function.

"We have written this function already. We just didn't call it a function. You see: we have already created these dialogues to ask the customer's name, the drink, flavor, and topping. Only one thing that we haven't done yet. Do you see it?"

"Return order?" Erik asked. "I see that we didn't do it, but I don't know what it means here."

"Look at the right side of the diagram. Here's what your order looks like, agreed?"

order:
 name: Erik
 drink: decaf
 flavor: vanilla
 topping: chocolate

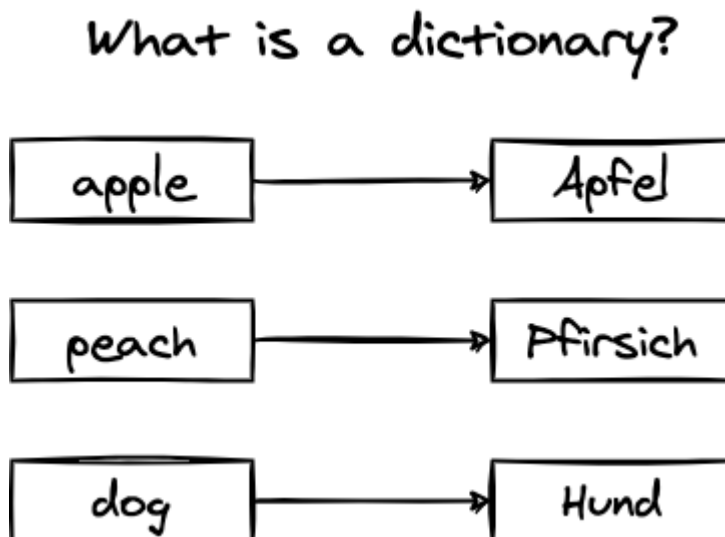
"We use the function `get_order()` to collect all this information, but instead of returning four separate values for name, drink, flavor, and topping, I want to return a single *thing* that I would call an order. And that single thing contains several values that go together as a whole."

"I know, you want to use a list here!" Erik shared his insight.

"That's one of the options, but I have in mind something better. In Python we have *dictionaries*. What is a normal dictionary?"

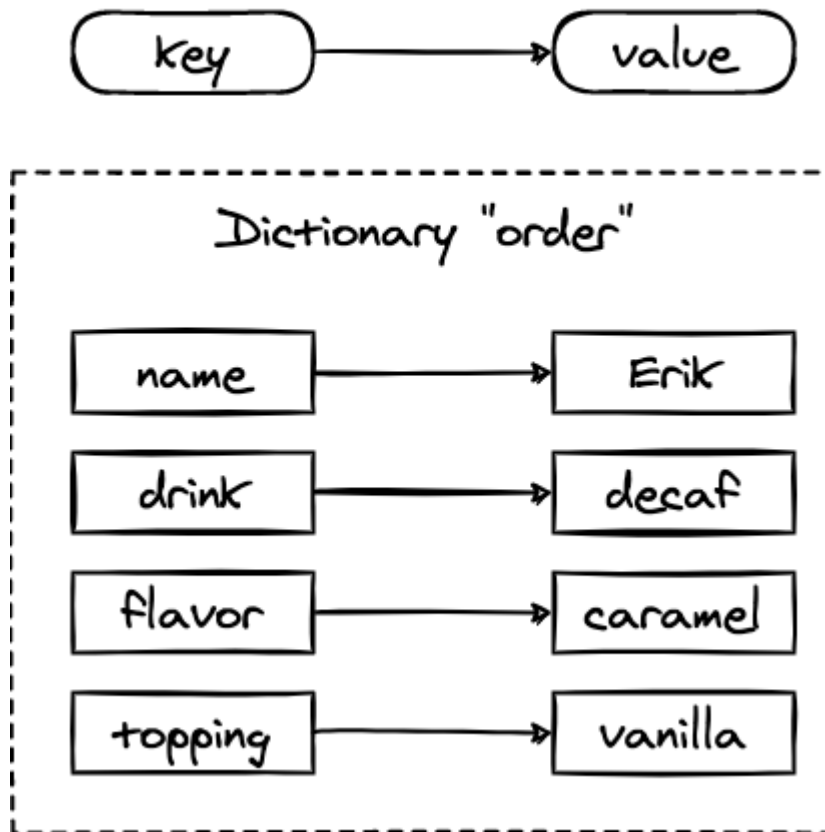
"Oh, it's a book with words and their meaning," Erik answered. "Or translations, if it's an English-German dictionary."

"Right!" Simon said. "You have a word and a value that is related to this word. Like this," and he draw a diagram.



"It could be its meaning or translation. In Python we have a similar thing. A dictionary in Python uses words that we call *keys* to get the *values* that are related to them. Let's look at your order. You have a key called 'name' and its value is 'Erik'. You have another key called 'drink' and its value is 'decaf'. And so on. The whole dictionary is called `order` and this is what we are going to return as a result from this function."

Dictionaries in Python:



"Let's practice with REPL again," Simon suggested. "Click REPL to get to the interactive session."

Erik clicked the REPL icon and switched to the window with an interactive session. It looked like this (your version might be different from the example below—that's okay):

```

Jupyter QtConsole 4.7.7
Python 3.8.12 (default, Mar 12 2022, 19:58:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.1.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
  
```

"We start with creating an empty dictionary called `order`. To create a dictionary in Python we use curly braces `{}` to make them look different from lists, that use square brackets `[]`. Type `order = {}` and then `ENTER`. This will be your order."

Erik typed:

```

In [1]: order = {}

In [2]:
  
```

"Now we can add items to your order. Let's start with the name. Type `order['name'] = 'Erik'`. Then try to print the order with a simple `print()` function."

Erik typed:

```
In [2]: order['name'] = 'Erik'

In [3]: print(order)
{'name': 'Erik'}

In [4]:
```

"But you said that dictionaries should use curly braces. Why do we use square brackets here?" Erik asked.

"Good question," Simon was so used to this Python feature that he couldn't find a good way to explain it right away. He started his answer, "Well, we use curly braces to *create* a dictionary. But we use square brackets to *access* the dictionary, when we want to get an item from it. In this sense it's similar to lists—when you want to get an item from a list you use square brackets. The difference is that with lists you use *indexes* that are integer numbers. With dictionaries you use *keys* that are usually strings. If you had a normal dictionary to get a word's meaning or translation, and you used its index—like 546—that would be very inconvenient, don't you agree? Instead, you use the word itself, like 'dog' and find is fast."

"Yes, right," Erik said. "Should I add the drink, flavor, and topping now?"

"Great idea, go ahead!" Simon said.

And Erik continued in his interactive session:

```
In [4]: order['drink'] = 'decaf'

In [5]: order['flavor'] = 'vanilla'

In [6]: order['topping'] = 'chocolate'

In [7]: print(order)
{'name': 'Erik', 'drink': 'decaf', 'flavor': 'vanilla', 'topping': 'chocolate'}

In [8]:
```

"Notice here," Simon said, "that your keys and values always go in pairs with a colon `:` between them."

YOUR TURN Learn dictionaries with REPL

Open REPL and work with dictionaries. You can repeat Erik's commands or create your own dictionaries. Try using different keys. Try to store a number instead of a string. Does it work?

"This is cool, I like it!" Erik said. "But can I print it in a better way, like I printed it before?"

"Of course," Simon said. "I think you should write a new function for that. But we'd better switch back to the editor for that."

"That will be my third function," Erik said.

"Are you still counting?" Simon smiled. "I'm sure pretty soon you'll lose count of the functions you've written!"

"Now let's get back to your editor and start writing the main menu program," Simon continued. We will use a *top-down* design approach here."

"What is it?" Erik asked.

"It's like what I just showed you: first we develop the algorithm for the whole program. We decide what are the big blocks and how we go from one to another. That usually includes decisions like confirm or cancel the order. We can develop the main program and use functions like `get_order()` or `print_order()`. It's not a problem if we don't have these functions yet. Before we write the real ones we can write very simple functions that would just print a message 'I am a function print_order()' and that's it. Some people call them 'placeholders'. When we see that the main menu works well and calls the right functions—then we'll add the real functions."

"Let me help you," and Simon took the keyboard. "First, I create a new file and save it as `main_menu.py`. Then I create a new *function* called `main_menu()` with the `def` keyword and parentheses."

"Another function?" Erik asked.

"Yes, in programming we usually create functions for everything. The main program is usually very short and it calls one of those functions. Then that function calls other functions and so on. So it's a good practice to write even your main menu as a function."

Simon continued, "Now look at the diagram. Do you see these arrows that go back to the 'Start'? They usually mean that in your algorithm you are going to *repeat* something. As soon as we return to the 'Start', we continue going through the same algorithm again and again. And to repeat something in a program we use... what?"

"A loop!" Erik answered.

"Exactly right!" Simon confirmed. "We used two types of loops already: for-loop and while-loop. Which one are you going to use here?"

"I think it should be a while-loop," Erik said. "This main menu looks similar to what we did in the drinks menu: repeat questions and check what the user answered."

"I agree," Simon said. "Look at your code where you wrote menu as a function. Remember, we used `while True:` there and checked what the user entered. What do you think we should check here? Hint: on diagrams these moments when we have to make a decision are usually drawn as rhombus."



"I see it!" Erik said. "It's where we ask the user if they want to confirm or cancel the order."

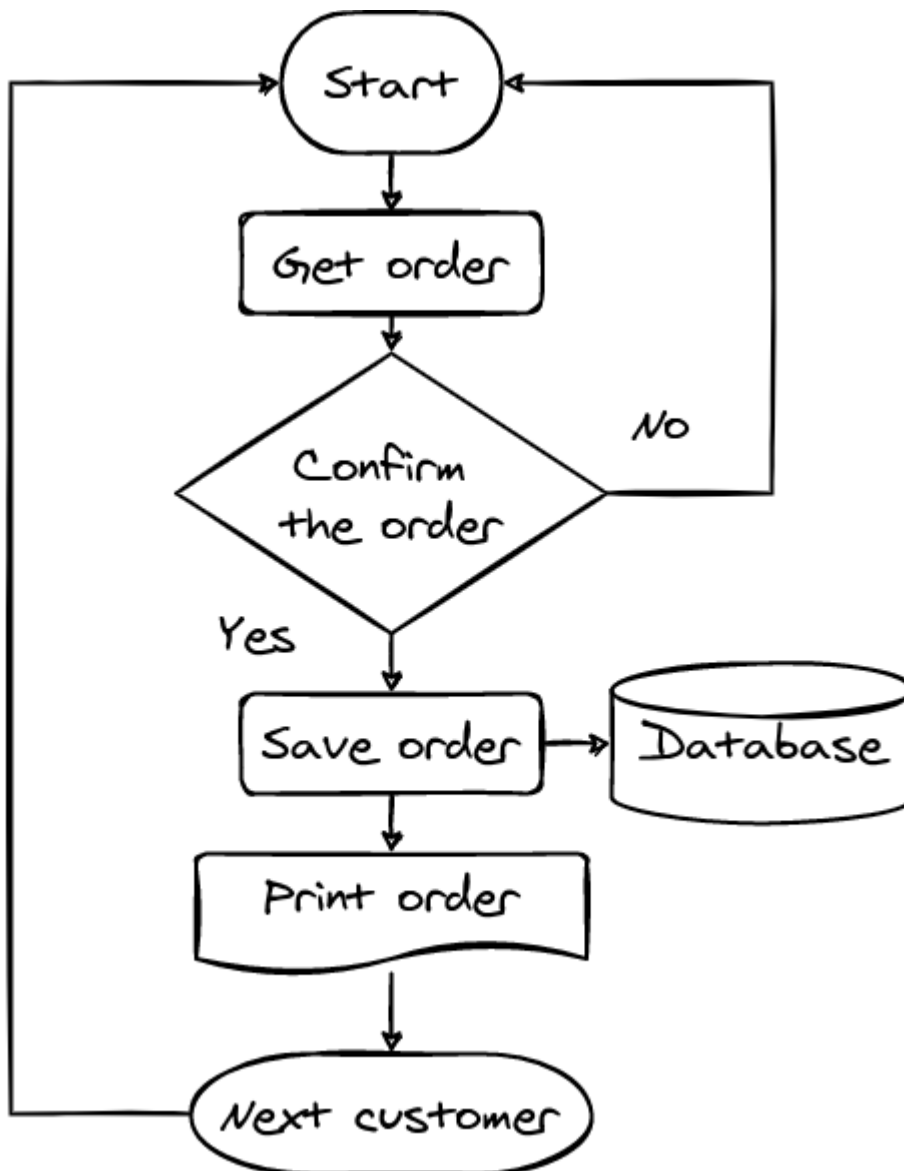
"Okay, let's start writing it," Simon suggested.

Erik wrote:

```
def main_menu():  
    while True:
```

"What's next?" he asked.

"Look at the diagram," Simon said.



"Get order?" Erik said.

"Right! And remember, that function `get_order()` will return a dictionary with the order. The dictionary will contain the customer's name, drink, flavor, and all that. We will put that dictionary into a variable called `order` in our main menu. Let's write this piece," and Simon added a line to Erik's code.

```
def main_menu():
    while True:
        order = get_order()
```

"What's next?" he asked his brother.

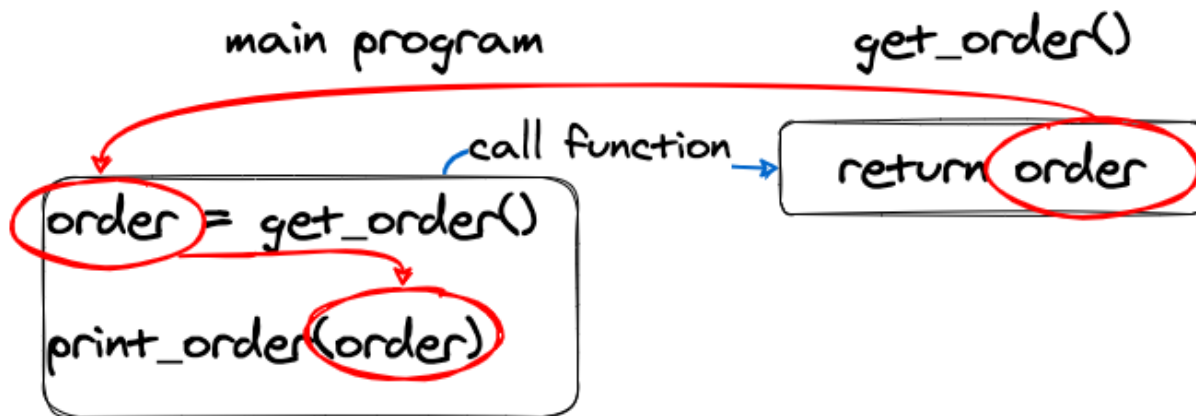
"Now we have to ask the customer if they confirm the order," Erik said.

"Good. But we have to show them the order before asking, I think," and Simon added a couple of

lines.

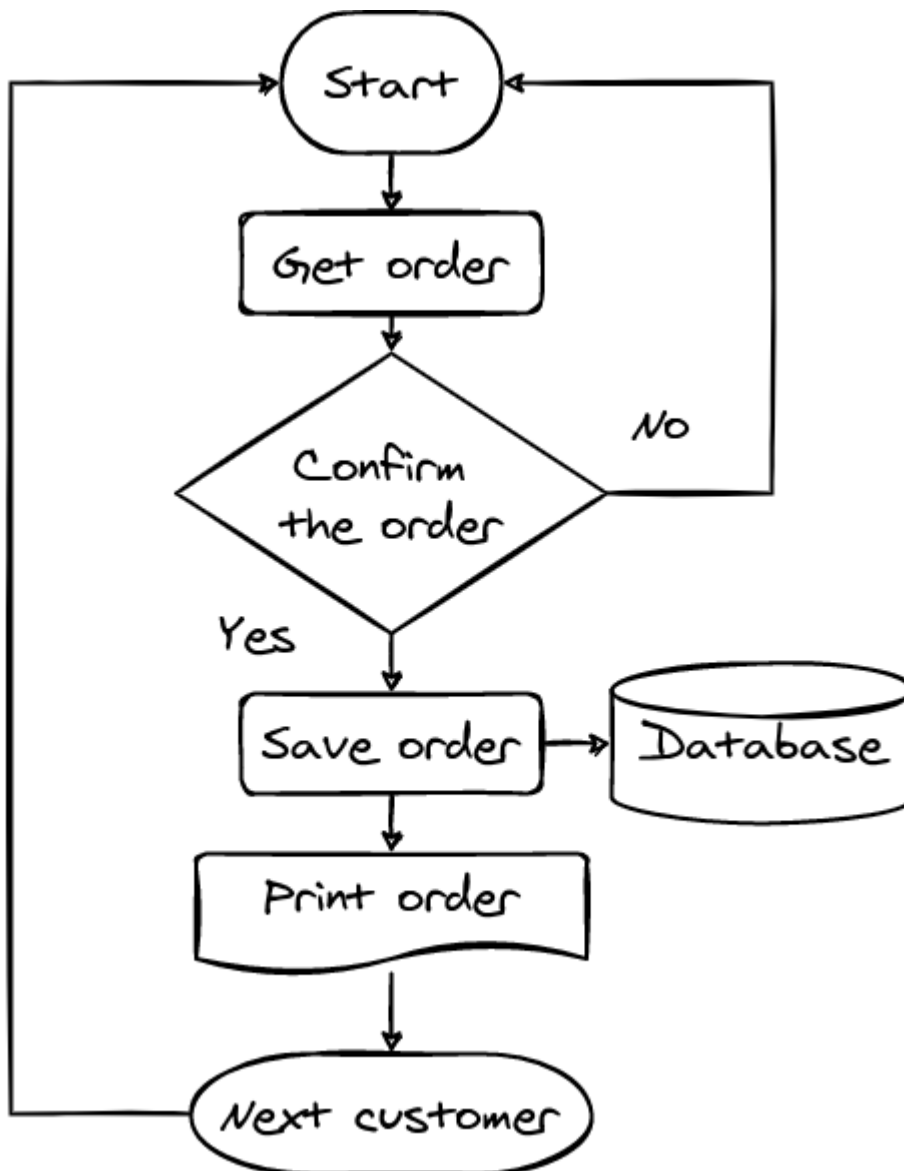
```
def main_menu():
    while True:
        order = get_order()
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel: ")
```

"Look here," Simon said to Erik, "I used the `order` variable that I received from `get_order()` as an argument for the next function, `print_order()`. This is very common in programming: we call one function to do something, it returns a result, and then we use that result as an input for another function."



"I see," Erik said. "Like in a movie theater: the cashier prints a ticket and gives it to you. Then you take the ticket, go to the entrance, you give the ticket to the guys at the entrance and they check it."

"Yes, good analogy, Erik! Let's continue it: we just received an answer to the question if the customer wants to confirm the order. Now, as you just said, we have to *check* the answer and decide what to do next. Like in a movie theater: they check if your ticket is correct and decide whether to let you in or not. Let's add these lines. Look at the diagram. If the user answers 'Yes', what shall we do?"



"Save the order and print the order," Erik answered looking at Simon's drawing.

"Okay, and if the user wants to cancel and responds 'No'?"

"We should do nothing, just return to the beginning. But I don't know how to do it. You just have an arrow here."

"There is a simple word for this arrow in Python and the word is `continue`. It means 'don't execute the rest of the loop and continue the loop from the beginning'. Pretty easy, huh?" and Simon added these lines to the function.

```
def main_menu():
    while True:
        order = get_order()
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel: ")
        if confirm == "Y" or confirm == "y":
            save_order(order)
            print("Thanks for your order:")
            print_order(order)
        else:
            continue
```

"I see that you've added two more functions: `save_order()` and `print_order()`," Erik said. "But we don't have them here..."

"Let's write them now!" Simon exclaimed. "We'll write very simple functions for now. They won't do anything, they will just print something like 'saving order...' so we will see that they were called. Later, we'll improve them to do more useful things."

Simon added the functions below the `main_menu()` function:

```
def get_order():
    return {}

def print_order(order):
    print(order)
    return

def save_order(order):
    print("Saving order...")
    return
```

He explained it to Erik: "The `get_order()` function is what you have written already. We will transfer your code here, but for now it does nothing. No menus, no dialogue, but it has to return the order. Remember, the order is a dictionary with keys like 'name', 'drink', and others. In this case the function returns just an empty dictionary, which is a pair of curly braces. So far, so good?"

"Yes," Erik answered. "So you mean we will copy my previous functions from that previous file into this one, right?"

"Right," Simon said. "Sometimes when a program becomes larger it's a good idea to group functions in separate files. But in our case it's easier to keep everything in one file."

"The `print_order()` function," Simon continued, "just prints the order that it gets via the argument. In this case we use the standard Python `print()`, but we will make it prettier later. You have done that already, remember?"

"Sure!" Erik said. "I think we can make it look like a real coffee shop receipt."

"Good idea," Simon said. "The `save_order()` function does nothing except printing 'Saving order...' That's okay for now, we'll write it later. Now we are ready to call the `main_menu()` function and test our algorithm. Go ahead and add the call for the `main_menu()` at the end and run it. Your main program will consist of only this function call."

Erik added the function call so the whole program now looked like this:

Listing 6.1 `main_menu.py`

```
def main_menu():
    while True:
        order = get_order()
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel: ")
        if confirm == "Y" or confirm == "y":
            save_order(order)
            print("Thanks for your order:")
            print_order(order)
        else:
            continue

def get_order():
    return {}

def print_order(order):
    print(order)
    return

def save_order(order):
    print("Saving order...")
    return

main_menu()
```

YOUR TURN Create main menu

Write a main menu function similar to the one Erik just created. Feel free to change the dialogue messages.

He clicked Run and saw the output:

```
Check your order:
{}
Confirm? Press Y to confirm, N to cancel:
```

He typed "y" and got this:

```
Saving order...
Thanks for your order:
{}
Check your order:
{}
Confirm? Press Y to confirm, N to cancel:
```

"Why does it give me the 'Check your order' again?" Erik asked.

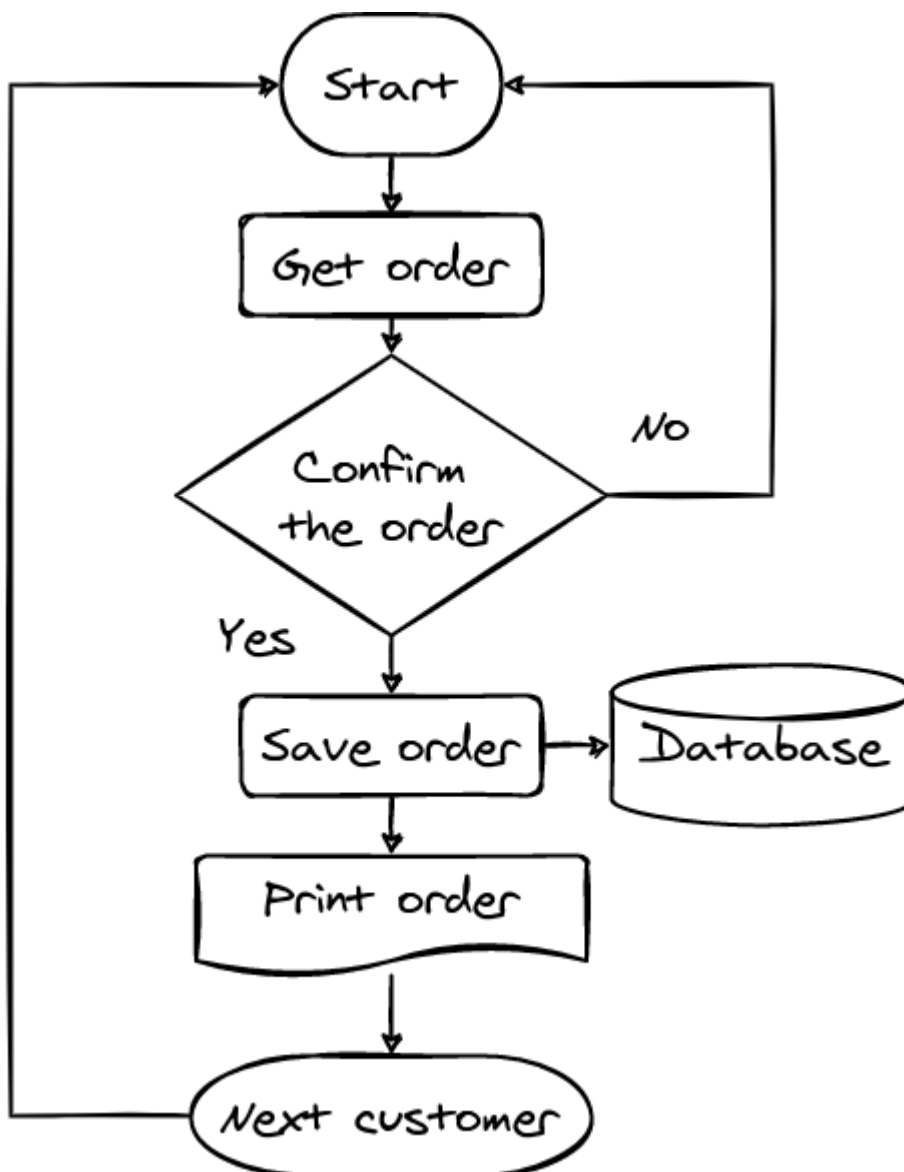
"Because it's a loop!" Simon said. "After you answered 'y' to the confirmation question it returns to the beginning of the loop. And as soon as we haven't added your menu dialogue yet, it prints the empty order. Everything works as expected. Now try to answer 'n' to the question."

Erik typed 'n' and got the output.

```
Confirm? Press Y to confirm, N to cancel: n
Check your order:
{}
Confirm? Press Y to confirm, N to cancel:
```

"Do you see the difference?" Simon asked.

"I see that it didn't print 'Saving order...' this time. That means it went on the short arrow on the right side of your diagram."



"Excellent!" Simon was glad to see that Erik really understood his algorithm diagram.

"I think we made a good progress today: our main menu is working. Tomorrow we'll write the actual functions that will do what we want. For now, let's wrap up and review what we learned today."

"We used the while-loop again!" Erik said.

"Right! And you used what you've learned while working on menus," Simon confirmed.

"Also we learned about dictionaries. They are like normal dictionaries, but you can store anything there, not just word descriptions."

"Yes, exactly! In our simple `order` dictionary we keep names, drinks, and flavors. But in a more complex dictionary you can keep numbers—prices, for example—and even lists and other dictionaries. dictionaries are really useful in Python and you will use them all the time."

"Also I liked how you created simple functions just to test the main menu," Erik said. "You said it's called 'top-down', right?"

"Exactly," Simon said. "There is also a 'bottom-up' approach, as you could guess. In that case people create functions first, test them properly, and then combine them into a large program. In some sense we used this approach too when you created and tested your first `menu()` function. Now we are going to use your function in our large program."

"Time to take some rest now," Simon continued. "We'll work tomorrow on the functions we used in our main menu."

6.1 New things you have learned today

- *Top-down approach*
First you develop the "big picture" of your application and use simple functions that just print something instead of doing real work. When the main algorithm works right you develop the actual functions.
- *Dictionary*
In Python dictionaries can store pairs of keys and values. You can assign values to keys and you can quickly find them by their keys.
- *Flowchart diagram symbols*
Programmers usually use diagrams to discuss their algorithms before they start writing code. Usually a rectangle means some process, a rhombus means a decision point with a Yes/No question. There are also symbols for input, using documents, using databases, and others. We will introduce them later.

6.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch06>



Creating functions: Get the order and print it

This chapter covers

- Erik creates the actual functions to get orders and print them
- Erik uses a dictionary to store and print a customer's order
- Erik's program now works as planned!
- Erik and Simon plan to write the function to save orders

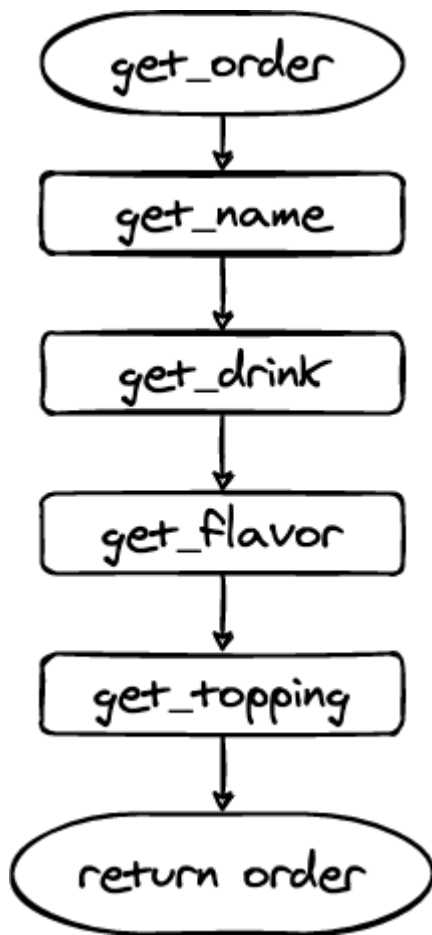
"Yesterday we created the main menu, right?" Simon began his conversation with Erik. "We even tested the main menu functionality."

"Yes, but it didn't do anything useful," Erik said.

"Right!" Simon said. "Remember, we talked about the 'top-down' approach? We created empty functions just to test the main menu. Now it's time to make them do something real. Open your Python file where we created menus from files. It's called `menu_files.py`."

Erik opened that file and now he had two tabs in his editor: one with the `main_menu.py` file and another with the `menu_files.py` file.

"Switch to the `main_menu.py` file and let's look which functions we have to write here," Simon said. "You see, three functions: `get_order()`, `print_order()`, and `save_order()`. Let's begin with `get_order()`. Where is our diagram?"



"Let's start with `get_name`. What do we do here?" Simon asked.

"We just ask 'what is your name?'," Erik answered.

"Right, and then?"

"And then we save it in a variable, like `name`."

"Almost right," Simon said. "Remember, we decided that `order` will be a dictionary. And we will save *everything* related to that order in this dictionary. For example, to save the customer's name instead of `name = 'Erik'` we should write `order['name'] = 'Erik'`. Only instead of 'Erik' we will use the function `input()` like you did in your first program."

"Let me try," Erik said and wrote this function:

```
def get_order():  
    order['name'] = input("What's your name: ")  
    return {}
```

"Now try it," Simon said.

Erik clicked Run, the program asked his name and he entered 'Erik'. But then he's got several

lines of error messages.

```
What's your name: Erik
Traceback (most recent call last):
  File "/home/erik/mu_code/main_menu.py", line 41, in <module>
    main_menu()
  File "/home/erik/mu_code/main_menu.py", line 3, in main_menu
    order = get_order()
  File "/home/erik/mu_code/main_menu.py", line 27, in get_order
    order['name'] = input("What's your name: ")
NameError: name 'order' is not defined
>>>
```

"What's that???" he asked Simon.

"Look, Python tells you where your problem is occurring. Read the last line."

"Name 'order' is not defined," Erik read it.

"It's very simple," Simon explained. "You tried to put something in the dictionary but you haven't created it yet. It's easy to fix. Let's create an empty dictionary. Remember, we used curly braces for that? Just write `order = {}` before the line with `input()`."

Erik changed his function to this:

```
def get_order():
    order = {}
    order['name'] = input("What's your name: ")
    return {}
```

and ran it again. This time it didn't give him any errors.

```
What's your name: Erik
Check your order:
{}
Confirm? Press Y to confirm, N to cancel: y
Saving order...
Thanks for your order:
{}
What's your name:
```

"It's better now," he said.

"Yes, better, but look: it still prints an empty order. You created an order and even entered your name into it, but your function returns an empty dictionary. See this line: `return {}`?"

"But this is how *you* wrote it!" Erik was sure it's not his fault.

"Yes, I wrote it this way to test the main menu function. But now we have to return the actual order dictionary. Change it to `return order` and let's see if it prints your name."

Erik changed the function to this:

```
def get_order():
    order = {}
    order['name'] = input("What's your name: ")
    return order
```

and ran it again. This time he saw:

```
What's your name: Erik
Check your order:
{'name': 'Erik'}
Confirm? Press Y to confirm, N to cancel:
```

"Yes, it prints my name now!"

"Congratulations!" Simon said. "Now you know how to work with dictionaries!"

YOUR TURN Create your `get_order()` function

Start writing your own `get_order()` function in the `main_menu.py` file. Add the first input to get the customer's name. Test it by running the main menu program.

7.1 What are your choices?

"Let's move on," Simon continued. "We have to add your `menu()` function now to fill drinks and flavors. But we also need the `read_menu()` function to read your menus from files. Copy both of them (`menu()` and `read_menu()`) from the `menu_files.py` file and paste them here in the `main_menu.py`. Paste them right before the `def get_order():` line."

"What if I paste it after that line?" Erik wanted to know why his older brother gave him such strict orders.

"Then it won't work," Simon gave him a simple answer and smiled. "Okay, if you *really* want to know: we are going to use these two functions in the `get_order()` function. First we have to read the menu contents from files: your drinks, your flavors, your toppings. Then we call the `menu()` function three times to get the customer choices. And before we can use these functions we should *define* them. In other words, we should tell Python that such functions exist and what they do. That's why they need to be pasted before the `def get_order():` line." By the way, this is why we use the word `def` to start a function—we *define* it."

"Okay," Erik said and started copying the functions. In a couple of moments his `main_menu.py` file looked like this:

Listing 7.1 main_menu.py

```
def main_menu():
    while True:
        order = get_order()
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel: ")
        if confirm == "Y" or confirm == "y":
            save_order(order)
            print("Thanks for your order:")
            print_order(order)
        else:
            continue

def menu(choices, title="Erik's Menu", prompt="Choose your item: "):
    print(title)
    print(len(title) * "-")
    i = 1
    for c in choices:
        print(i, c)
        i = i + 1
    while True:
        choice = input(prompt)
        allowed_answers = []
        for a in range(1, len(choices) + 1):
            allowed_answers.append(str(a))

        allowed_answers.append("X")
        allowed_answers.append("x")

        if choice in allowed_answers:
            if choice == "X" or choice == "x":
                answer = ""
                break
            else:
                answer = choices[int(choice) - 1]
                break
        else:
            print("Enter number from 1 to ", len(choices))
            answer = ""
    return answer

def read_menu(filename):
    f = open(filename)
    temp = f.readlines()
    result = []
    for item in temp:
        new_item = item.strip()
        result.append(new_item)
    return result

def get_order():
    order = {}
    order["name"] = input("What's your name: ")
    return order

def print_order(order):
    print(order)
    return

def save_order(order):
    print("Saving order...")
    return

main_menu()
```

"Correct!" Simon said. "Now you know the rule: define something before using it. You just saw the problem when you got errors with the `order` dictionary, and now you see it with the `menu()` and `read_menu()` functions here."

"Now we are ready to use these functions in `get_order()`," he continued. "Look at your `menu_files.py` program. What did we do first?"

"We read the menus from files," Erik answered.

"Good, let's do it here, but inside the function."

Erik added three lines to the `get_order()` function:

```
def get_order():
    order = {}
    order["name"] = input("What's your name: ")
    drinks = read_menu("drinks.txt")
    flavors = read_menu("flavors.txt")
    toppings = read_menu("toppings.txt")
    return order
```

He had to adjust the lines by adding four spaces before each line so they were all indented at the same level.

"And now the same with three `menu()` functions?" he asked Simon.

"Sure, go ahead!"

Erik changed his function to this:

```
def get_order():
    order = {}
    order["name"] = input("What's your name: ")
    drinks = read_menu("drinks.txt")
    flavors = read_menu("flavors.txt")
    toppings = read_menu("toppings.txt")
    drink = menu(drinks, "Erik's drinks", "Choose your drink: ")
    flavor = menu(flavors, "Erik's flavors", "Choose your flavor: ")
    topping = menu(toppings, "Erik's toppings", "Choose your topping: ")
    return order
```

He was proud of his work and looked at Simon.

"Almost right," Simon said. "You copied it right, but you have to change the code a bit to store the answers in the `order` dictionary. It should be an easy change, you know how to do it."

"Ah, I see," Erik said and changed the function. Now the function looked like this:

Listing 7.2 main_menu.py

```
def get_order():
    order = {}
    order["name"] = input("What's your name: ")
    drinks = read_menu("drinks.txt")
    flavors = read_menu("flavors.txt")
    toppings = read_menu("toppings.txt")
    order["drink"] = menu(drinks, "Erik's drinks", "Choose your drink: ")
    order["flavor"] = menu(flavors, "Erik's flavors", "Choose your flavor: ")
    order["topping"] = menu(toppings, "Erik's toppings", "Choose your topping: ")
    return order
```

Simon encouraged him: "Go ahead, run it!"

Erik ran the program.

```
What's your name: Erik
Erik's drinks
-----
1 coffee
2 chocolate
3 decaf
Choose your drink: 1
Erik's flavors
-----
1 caramel
2 vanilla
3 peppermint
4 raspberry
5 plain
Choose your flavor: 2
Erik's toppings
-----
1 chocolate
2 cinnamon
3 caramel
4 vanilla powder
Choose your topping: 3
Check your order:
{'name': 'Erik', 'drink': 'coffee', 'flavor': 'vanilla', 'topping': 'caramel'}
Confirm? Press Y to confirm, N to cancel: y
Saving order...
Thanks for your order:
{'name': 'Erik', 'drink': 'coffee', 'flavor': 'vanilla', 'topping': 'caramel'}
What's your name:
```

"Wow!" he was really happy. "I wrote a program of more than 70 lines and it works!"

"Yes, you did! And it really works!" Simon confirmed and smiled.

YOUR TURN Add menu choices to your program

Add the two functions `menu()` and `read_menu()` like you see in the preceding program to your file `main_menu.py`. Test the program by running it and entering your choices. Try entering wrong choices and make sure the `menu()` function doesn't allow you to do it.

7.2 Print it!

"But something is still missing... The order doesn't look very professional. It doesn't look like a real coffee shop..." Simon continued.

"I see, it should be in the `print_order()` function, right?" Erik suggested.

"Yes, right. Go to the `print_order()` function in your `main_menu.py` file. Let's start printing the order."

Erik's `print_order()` function looked like this:

```
def print_order(order):
    print(order)
    return
```

"Here we use the default printing function provided by Python," Simon continued. "Python *can* print your dictionary, but it's not pretty. It's okay for debugging, but for real orders and receipts we have to make it more beautiful. And you have done that already, right?"

"You mean—when I printed lines of dashes? Yes, it was prettier than this."

"Okay, let's do something similar to what you did at the end of the `menu_files.py` file. You can copy those lines starting with `print` from there. Just don't forget to keep the right indentation and make sure you use the dictionary and not simple variables. Ah, and don't forget that now we have the customer's name. I think you should use it in your function. Ready?"

"Yes," Erik answered and started working on the function. He ended up with this:

```
def print_order(order):
    print("Here is your order, ", order["name"])
    print("Main product: ", order["drink"])
    print("Flavor: ", order["flavor"])
    print("Topping: ", order["topping"])
    print("Thanks for your order!")
    return
```

YOUR TURN Add the `print_order()` function

Add the `print_order()` function to the file `main_menu.py`. Feel free to use decorations like dashes (`-`), underscores (`_`), or equal signs (`=`) to make your printed order look like the ones you saw somewhere else. Try to find receipts from restaurants, coffee shops, ice cream shops and see if you can make yours look similar.

He ran the program again and got much prettier output:

```

Here is your order, Erik
Main product: coffee
Flavor: vanilla
Topping: caramel
Thanks for your order!
Confirm? Press Y to confirm, N to cancel:

```

"Yes, this is much better!" Simon said. "Yes, you can add decorations like dashes and vertical lines, it's up to you. But I can tell you that you did a great job as a programmer. You wrote several very useful functions, you organized them properly, and you tested them. Good job, Erik, I'm really proud of you!"

"Also we learned about dictionaries and I used them," Erik sensed that it's a "wrap-up time" and he should mention everything he learned and used today.

"Yes, right," Simon confirmed. "Dictionaries are very important in Python. We use them all the time in our programs. Later we'll learn more about them."

"You said we should also save our orders somehow. Will we do it tomorrow?" Erik asked.

"Yes, sure," Simon said. "Do you know about JSON?" he asked.

"Jason? Yes, we go together to our math class. What about him?"

"No, not that Jason," Simon laughed. "JSON is a file format that we can use to save your coffee shop orders. We'll learn about it tomorrow, okay?"

"Okay," Erik said and off he went.

7.3 New things you have learned today

- *Variable and function definitions*

In Python we have to *define* variables and functions before we can use them. For variables it is as simple as assigning an empty value to it. For a dictionary it is: `order = {}`, for a string it is: `name = ""`. Functions should be defined using the keyword `def`.

7.4 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch07>

Working with JSON: Save the order

This chapter covers

- Erik learns about JSON format and files
- Erik learns about Python modules
- Erik creates a list of orders
- Simon and Erik write the function to save orders in a JSON file

"You said something about Jason yesterday," Erik asked Simon. "But you said it's another Jason, not the one from my math class."

"Yes, that's another Jason," Simon smiled. "It's JSON, J-S-O-N, the file format we use to store data."

"Like files we used to store menus?"

"Yes, similar," Simon answered. "This format is very good to store *structured* data."

"What is that?" Erik asked.

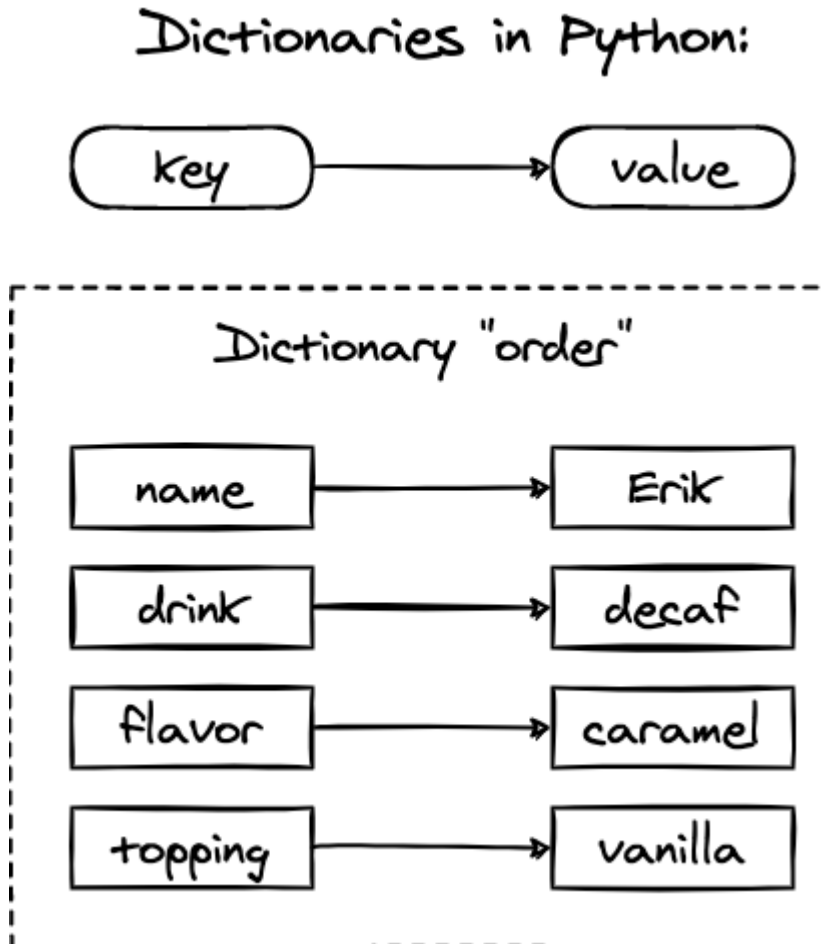
"Sometimes you want to store just a piece of text, or an image. Usually, they don't have any fixed structure. A text is just a text. An image can be large and small, it can be black-and-white or it can have color. But it doesn't have any structure—it's just a bunch of pixels. This is what is called *unstructured* data. But in your case, each order has a *structure*. Each order has the customer's name and all the components of the drink you are going to prepare. No more, no less. It always has the main drink, the flavor, the topping. Because of the menu you wrote, the customer should answer all these questions before you can print or save the order. On the other hand, the customer can't *add* anything to the order."

"Like another topping?" Erik asked.

"Yes," Simon said. "Your order is an example of *structured* data."

Erik didn't think that he just created something with such a serious name.

"Your order is stored in a dictionary," Simon continued, "and you know for sure that for each order, there are dictionary *keys* (remember what that is?) called 'name', 'drink', 'flavor', 'topping'." Simon pulled one of his diagrams.



"The JSON format is created to store this kind of structured data. Let's practice with it a little. Like we did before, we'll create a simple program first, and try some simple operations. Then, as previously, we'll use what we learned with this simple program and make our main program save orders in a file. That's a lot for one day, so maybe we'll do it tomorrow."

Simon continued: "Now open your editor and create a new file. Save it with the name `dict_json.py`, for example."

Erik opened his editor window, clicked New, then Save, entered `dict_json.py`, and clicked Save again. He was already familiar with the procedure.

"Now," Simon said, "create a sample order."

"What is a sample order?"

"Your order is a dictionary, right?" Simon started to explain. "In your main program you created an empty dictionary and then started to fill it with the values you were getting from the customer. Here we want to skip that step and imagine that our `order` dictionary is already filled with the customer's choices. Let me start it for you," Simon said and wrote in Erik's editor:

```
order = {
    "name": "Erik",
```

"You can continue now," he said. "Don't forget to close the curly braces."

Erik finished the order dictionary and closed the curly braces. Now it looked like this:

```
order = {
    "name": "Erik",
    "drink": "coffee",
    "flavor": "caramel",
    "topping": "chocolate"
}
```

"I noticed that you indented the lines in this dictionary," he asked his brother. "Is it also a rule for dictionaries in Python?"

"No," Simon answered, "in this case I did it just to make it look better. And to be more *readable*. I could put all items together in one line, or start from the beginning of the line, but I think this way it looks better."

"Now," he continued, "we have a dictionary. And we want to save it in a file. I guess I should remind you about files operations with dictionaries."

"Yes," Erik said, "it was so-o-o long ago, I don't remember much."

"Sure," Simon said. "Also you'll learn a couple of new things about files. First, we have to *open* a file. To open a file we should call a function named `open()`--of course!--and pass the file name as an argument. You know everything about functions and their arguments, right? That function returns a file *handle*. It's a special object that our program can now use to work with this file."

Simon wrote one more line below the dictionary that Erik created.

```
order = {
    "name": "Erik",
    "drink": "coffee",
    "flavor": "caramel",
    "topping": "chocolate"
}

f = open("orders.json", "w")
```

"Here is the first new thing. See this 'w' letter? It means that we are going to write into the file."

```
f = open("orders.json", "w")
```

"But when we opened the menu files we didn't use any letters," Erik remembered.

"You are right! Good thing you remembered it," Simon said. "Yes, we didn't use any letters—they are called *modes*, by the way—because by default, when I don't use any letters, Python opens files for reading. This time we want to write to this file so we tell Python about it."

"And I see that you named the file 'orders.json'. Is it because you want to use that JSON format you were talking about?"

"Yes, exactly," Simon answered. "It's not mandatory, but it's a convention to add the '.json' extension to JSON files. Another difference is that when we use the 'write' mode, Python will create the file with this name if it doesn't exist."

"What now?" Erik asked. "How do we write to this JSON file? Last time we used 'methods'; is this what they are called?"

"Yes, you remembered it correctly," Simon said. "But this time we'll do it differently. It's all because we are going to write structured data, not just plain text. We are going to use a Python *module* called `json`."

"What is a module?" Erik asked immediately.

"I'm going to explain it right now," Simon smiled. "Remember, you wrote several Python functions recently? For example, to read menu items from a file and return a list. Imagine one of your friends wants to write their own program for a coffee shop or something similar."

"Yes," Erik said, "I spoke with Emily recently and she said she wanted to create a program for an ice cream shop."

"Great!" Simon said. "Now you may want to help her and share the functions that you wrote. It will save her some time so her program will be ready earlier. It's very common among programmers to share their work to help each other. In Python you can group your functions that you want to share in a file and give it to Emily. She can copy that file to her computer and then *import* it into her program. Because she imported it she can use your functions in her application. Now your file with functions is called a *module*."

"What if I don't want to share my functions?" Erik said. "I spent several days writing them!"

"Yes, you did," Simon said. "And you did a great job. But remember, a lot of people spent many days writing other functions in Python, and even Python itself. And they shared their work with other programmers so you can use Python and other functions completely free. That way we help

each other to work on our projects. It would be much slower if you and I had to write everything ourselves from scratch. That's why people use somebody else's code and also share their code with others. It's usually called the *Open Source community*."

"Back to JSON," Simon continued. "We are going to use the module called `json`, written by other people. That module can read Python dictionaries and convert them to JSON files. Go to the beginning of your file and add a line: `import json`. It should be the very first line of the file."

Erik did. Here is his updated file.

```
import json

order = {
    "name": "Erik",
    "drink": "coffee",
    "flavor": "caramel",
    "topping": "chocolate"
}

f = open("orders.json", "w")
```

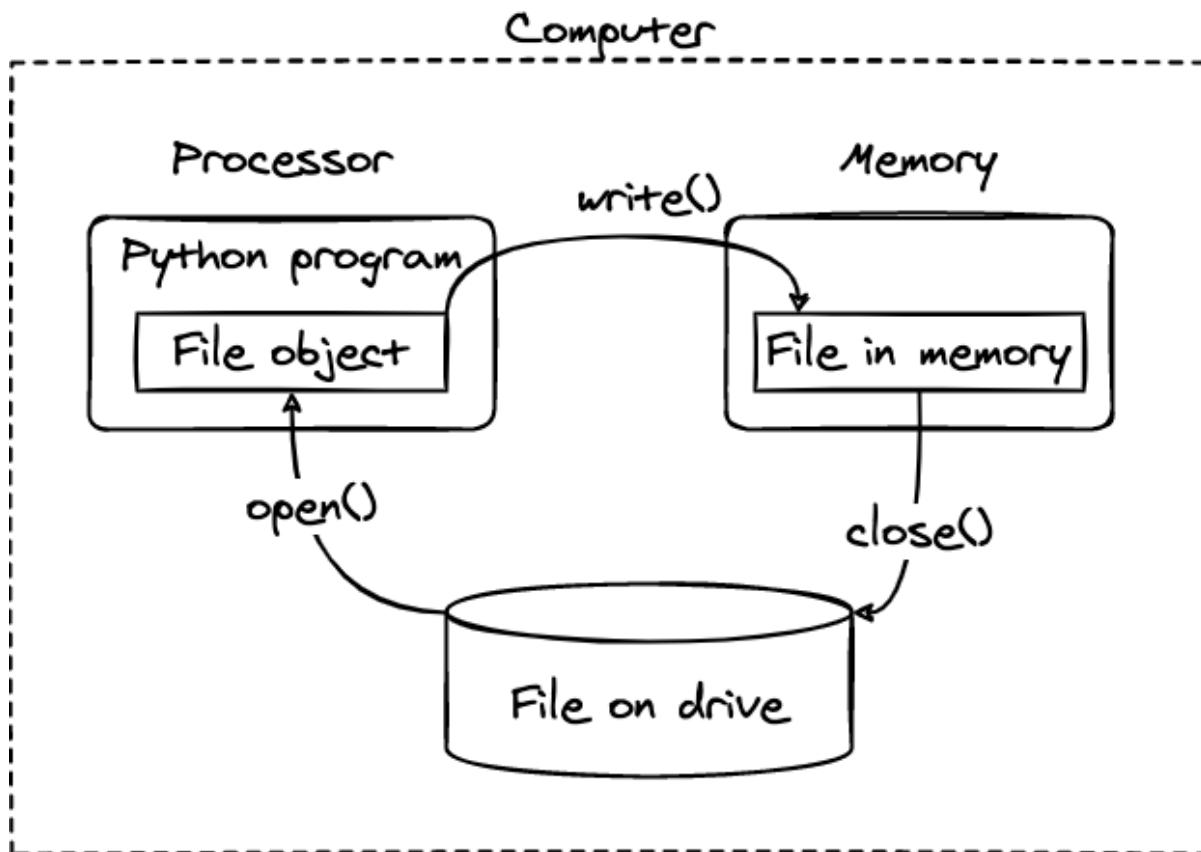
"Now we have to convert your example dictionary to JSON and write it to the file we just opened," Simon said. "In the `json` module, this function is called `dump`. We call it in your program, but we have to tell Python that it should look for this function in the module `json`. So we call it like this: `json.dump()`. You just have to pass two arguments: the dictionary and the file object. Add this function to the end of the file. Your dictionary is `order`, your file object is `f`."

Erik added this line to the end of the program:

```
json.dump(order, f)
```

Simon continued: "Now here is another thing that we didn't do with files before. We should *close* the file. This is important, so let's draw another diagram."

"Look, here are three main component of a computer: the processor, the memory, and the drive. Your Python program is running on the processor. Your file is stored on the drive in a file system. A file system is what you see in Finder on a Mac, and in Explorer on Windows: folders and files. When you want to work with a file in Python you *open* it, like you just did. That creates a file object in your program. When you write to the file, you write to the computer memory. Then when you want your file to be *really* written to the file system on the drive, you *close* it."



"This is so complicated!" Erik was confused. "Why don't we write straight to the drive?"

"Computers *are* complicated, you are right!" Simon agreed. "The reason for that is that computer engineers try to make computers work faster. Writing to drive is slow—much slower than writing to memory. Imagine you are writing your program in a text editor. If it saved every letter you type immediately to drive it would be *very* slow. You don't like working with slow computers, do you?"

"I hate slow computers!" Erik answered emotionally.

"To make computers work faster," Simon continued, "engineers decided to store data in memory and save it to the drive only when necessary. There are a lot of tricks they use to make computers run faster, and, of course, I don't know all of them. Maybe I'll learn more about them in college."

"Let's get back to our program. Remember: file objects use methods. In this case we call `f.close()`. That will make sure our order is written to the file. Now add it after the last line in your program."

That was easy—after such a long explanation! Erik added it quickly and got this.

```
import json

order = {
    "name": "Erik",
    "drink": "coffee",
    "flavor": "caramel",
    "topping": "chocolate"
}

f = open("orders.json", "w")
json.dump(order, f)
f.close()
```

"Now run it," Simon said.

Erik clicked Run and saw the familiar `>>>` at the bottom of the window. "Now what?" he asked Simon.

"Nothing happened?" Simon smiled, feeling Erik's confusion. "Of course, because you didn't tell Python to print anything. But still, something happened behind the scenes. Python opened a file called `orders.json`, wrote your dictionary into it, and closed it. Now we have to open it to check if it did it right. Use a plain text editor to open the file. You are on a Mac so it will be *TextEdit* from your Applications folder. On Windows it's *Notepad*, on Linux it's *gedit* or *Kate*. Start the editor and open the file. It's in your home folder, under `mu_code` and it's called `orders.json`."

Erik started *TextEdit*, found the file, and opened it. Indeed, he saw his order:

A screenshot of a text editor window. The title bar shows three colored window control buttons (red, yellow, green) on the left and the filename "orders.json" in the center. The main text area contains a single line of JSON text: {"name": "Erik", "drink": "coffee", "flavor": "caramel", "topping": "chocolate"}.

```
{ "name": "Erik", "drink": "coffee", "flavor": "caramel", "topping": "chocolate" }
```

YOUR TURN Save your example order in a JSON file

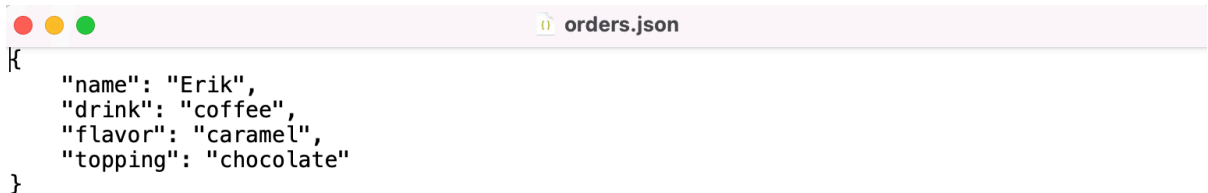
Write the program Erik just wrote. Try to use a slightly different order. Run the program and check the resulting JSON file with a text editor on your platform. Try to change the order and run the program again. Did your JSON file change? (You may have to reload the file in your text editor.)

"You see," Simon said, "it's your sample order, stored in a file. Let me add something and you will see why JSON files are ideal for storing Python dictionaries."

Simon took the keyboard and changed the `json.dump()` call to this:

```
json.dump(order, f, indent=4)
```

He ran the script again and re-opened the `orders.json` file. Now it looked like this:



```
{
  "name": "Erik",
  "drink": "coffee",
  "flavor": "caramel",
  "topping": "chocolate"
}
```

YOUR TURN Make it beautiful

Add the `indent=4` argument to your previous program and check if your JSON file has changed.

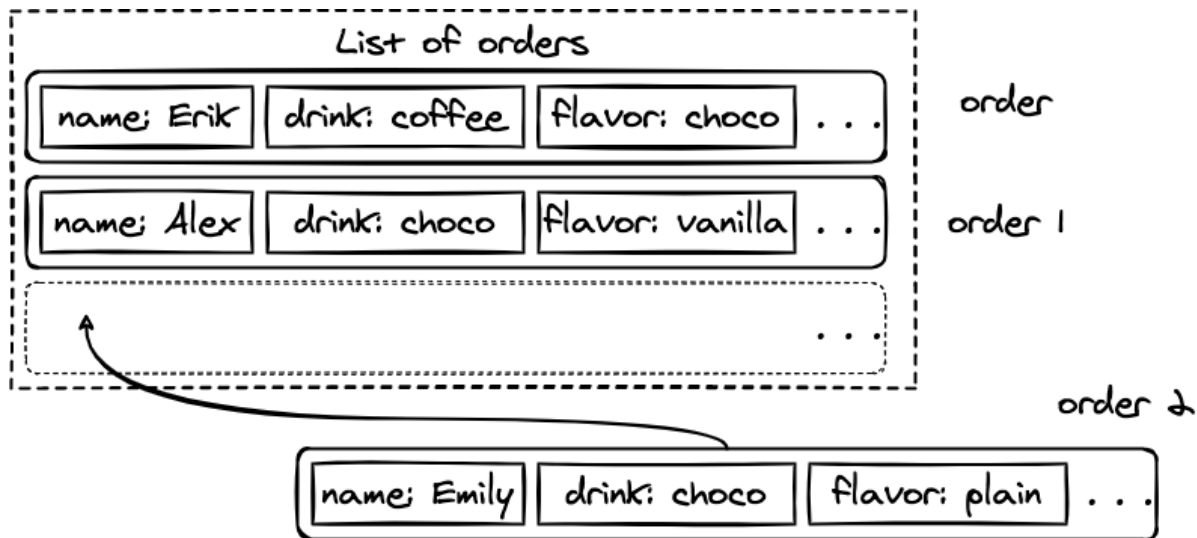
"It looks exactly like my dictionary!" Erik said.

"I told you!" Simon exclaimed. "We'll be using JSON to store your orders. I said 'orders,' which

means now we have to learn how to keep several orders and store them in a file. We know already how Python stores several items that come in order—you used it for you menus."

"A list!" Erik said.

"Correct! A list in Python can contain different things: strings, numbers, even dictionaries. In this case we'll have a list of dictionaries. Each dictionary will contain an order we'll add them one by one to the list. A new customer, a new order, a new dictionary in the list. Let me draw a diagram."



"Let's create a list of orders," Simon continued. "Copy the existing order in the code and call it `order1`, for example.

Then change the order's content: the name, the drink, and others."

Erik worked on his code for a while and finally got this additional order, just below the first one.

```
order1 = {
    "name": "Alex",
    "drink": "choco",
    "flavor": "vanilla",
    "topping": "caramel"
}
```

"Good," Simon said, "now create an empty list called `orders`. Note that it's plural—'orders'. It's very similar to creating an empty dictionary—you have done that already. Just instead of curly braces use square brackets."

Erik added the following line below the second order:

```
orders = []
```

"And now we will add both orders to the list," Simon said. "Believe it or not, but the list `orders`

that you just created is also an object. In Python, actually, everything is an object. And each object has methods that you can use. You just have to know what methods exist for each object. For example, for all lists there is a method called `append()`. It adds the element you pass as an argument to the end of the list. Look here, I'll use it to add `order` and `order1` to the list `orders`." And Simon added these two lines below the line where the list `orders` was created.

```
orders.append(order)
orders.append(order1)
```

"But how do you know that you should use `append()` here?" Erik asked.

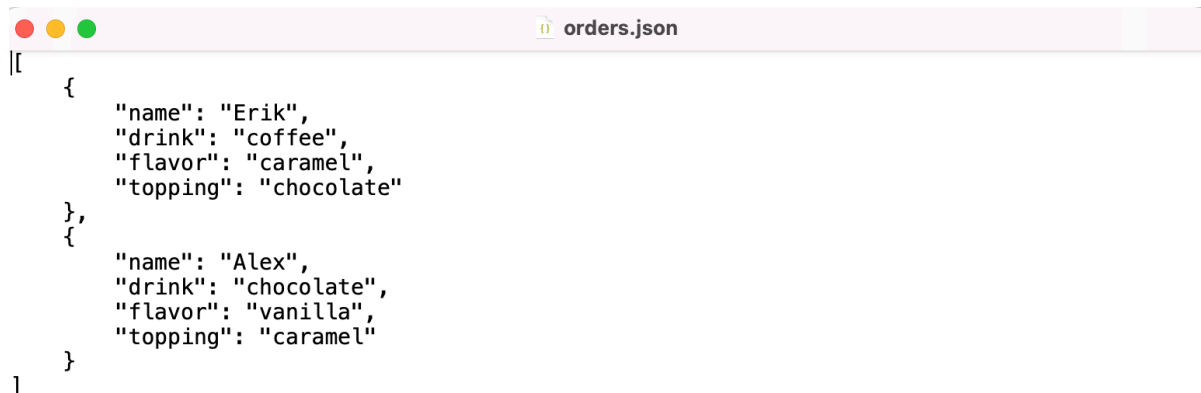
"I read it in Python documentation when I learned Python," Simon answered with a smile. "You can use Google to search for 'Python list methods' and you'll find everything you need to know."

"Now we can try to save this list as JSON," Simon continued. "The only thing we need to change is *what* we want to 'dump' into the file. We used `order` as an argument for the `dump()` function to write out one order. Now let's replace it with `orders` (plural!) and see what changes."

Erik changed the line with `json.dump()` to this:

```
json.dump(orders, f, indent=4)
```

He ran the program and opened the `orders.json` file again.



```
[
  {
    "name": "Erik",
    "drink": "coffee",
    "flavor": "caramel",
    "topping": "chocolate"
  },
  {
    "name": "Alex",
    "drink": "chocolate",
    "flavor": "vanilla",
    "topping": "caramel"
  }
]
```

YOUR TURN Save a list

Add another example order. Call it `order2`. Create a list of orders. Save the list in the same JSON file. Check the result with a text editor. Add as more orders as you can think of and write them out to the file. Is there any limit to how many orders you can save?

"So, what would you say?" Simon asked. "Does it look like your orders?"

"Yes, it's exactly like Python!" Erik said. "But why did we write my orders in a separate file? If it looks like Python, why don't we write my orders into our Python program?"

"Great question!" Simon was really glad that Erik wanted to understand things. "First of all, we always want to separate programs from data. Remember, when you run your Word application, you don't write your documents into the Word program. You save them in separate files. It's exactly what we do here. Your program can save orders in different files, for example, for different days. All you would have to do is change the name of the output file such as `orders.Monday.json`, `orders.Tuesday.json`, etc.

"Second reason," Simon continued, "is that this format called JSON for a reason. It stands for JavaScript Object Notation. First it was invented by people who used the JavaScript programming language, and then other languages started using it. So you can use Python to write your orders in a JSON file, and then some of your friends may want to create another program in JavaScript that would read from that file and print your orders on the web page, for example."

"Yes, I heard some people in my class said they know JavaScript!" Erik said.

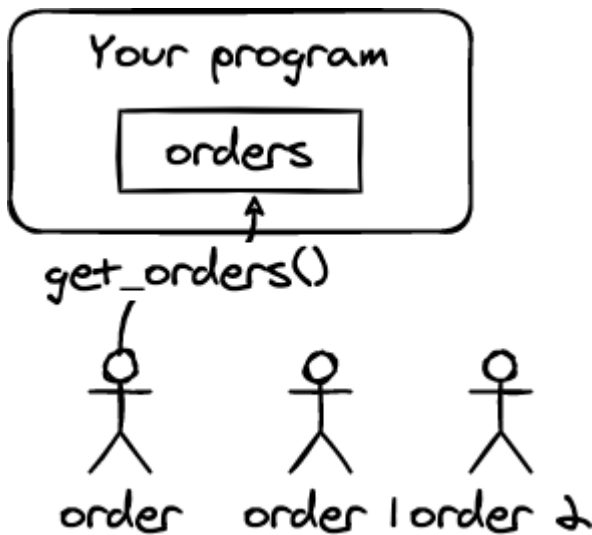
"Good! You may want to create a programming team and work on applications together," Simon said. "But let's continue with our sample program. Now we will read the orders from the JSON file and save them into a new list. Let's call it `saved_orders`."

"Why are we reading it if we just have written it to the file?" Erik was confused.

"Maybe I didn't explain it properly," Simon answered. "In this sample program we are practicing some operations with JSON files, so we know them well and can use in our main program. Programmers do this very often: they create simple programs to test some concepts and ideas. Let me show you my plan for our main program so you better understand where we are going."

Simon took another piece of paper and started drawing.

"First, we check if the file called 'orders.json' exists. If it exists, we open it and read from it our previous orders."

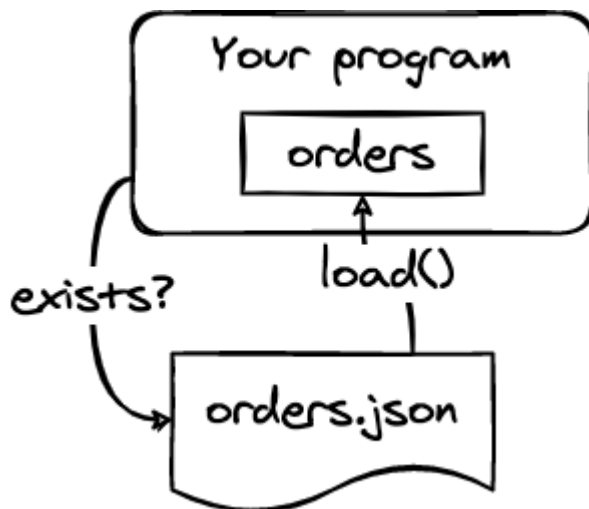


"Why do we need our previous orders? We have prepared them already," Erik asked.

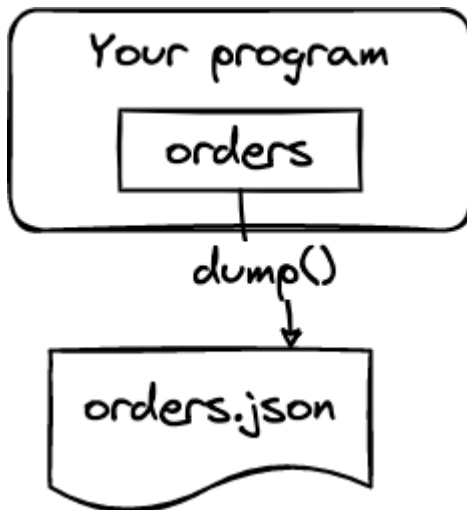
"Yes, but remember, we may want to count how many customers we served today or yesterday or last month. Or count how many portions of caramel we have used and if it's time to buy more. You need all the orders if you want to manage your coffee shop business seriously. That's why all businesses keep these records for a long period of time."

"What if we don't have this file?" Erik asked.

"That means we just have opened our business and started working," Simon smiled. "In this case we create an empty list and start getting orders. The file will be created automatically when we open it for writing."



"Look at this diagram: here we have our 'orders' list either filled with our previous days orders or empty. And we start getting orders and save them into this list. After we are done for the day, we close the file and that saves all our orders on the drive. Next day we open the file again and continue taking orders. All the new orders will be added to the previous day's orders."

**YOUR TURN** Draw your own diagrams

Try to draw the diagrams for your program without looking in the book. Drawing diagrams helps you to understand how programs work.

"Is this how real coffee shops work? Like *Starbucks*?" Erik asked.

"Yes, pretty much," Simon smiled. "Of course, they use a database for reliability and security. Their order records are more complex than ours. But the whole process is very similar."

"Now when you know the grand plan, let's continue with our simple program and read from the file. We will read the previous orders into a new list called 'saved_orders' and then we'll just print it to see if we read it correctly. To do that, in the `json` module there is a function called `load()`. It works the same way as `dump()`: first we open a file, but this time for reading, not writing. Then we call `json.load()` and pass the file object as an argument. The function *returns* the object it read from the file and we assign that object to a variable. In our case it will be a list of orders—that are dictionaries, as you remember. Sounds complicated? Let me help you. It's much shorter in Python," and Simon started adding lines to Erik's code. Here is what he added at the end of the program:

```
f = open("orders.json", "r")
saved_orders = json.load(f)
print(saved_orders)
```

He clicked Run and Erik saw the output:

Running: dict_json.py

```
[{'name': 'Erik', 'drink': 'coffee', 'flavor': 'caramel', 'topping': 'chocolate'}, {'name': 'Alex', 'drink': 'chocolate', 'flavor': 'vanilla', 'topping': 'caramel'}]
>>> |
```

YOUR TURN Read from the JSON file

Add the preceding lines to your program and try to read from the JSON file you have created. Do you get the same orders as in your example orders?

"We learned a lot today," Simon said. "Let's take a break until tomorrow. Tomorrow we'll add these functions to our main program and then it'll become a real coffee shop application. Can you quickly recap what we did today?"

"We created a JSON file from my Python dictionary and it looked very much like Python. Then you explained me all about files, and memory, and drives. And also we created a list of dictionaries and saved it in the file too."

"And also we learned about Python modules and how to import them," Simon added. "So far, so good," he said, "Let's continue tomorrow. We are very close to finishing the first version of your application."

8.1 New things you have learned today

- *JSON, JavaScript Object Notation*
A format that is used to store structured data and can be used to exchange information between programs.
- *Python modules*
Groups of Python functions that can be used by other programmers. Usually they are grouped in files. You have to `import` modules before using them.
- *List of dictionaries*
Lists can contain different types: strings, numbers, dictionaries, other lists.
- *File operations*
You can open files for reading and writing. You can write data to files, but it's written in computer's memory. You should close files to save the data onto the computer drive.
- *Open Source Community*
People who share programs they write and help each other write better code.

8.2 Code for this chapter

You can find the code for this chapter here:
<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch08>

Complete the menu: A real program

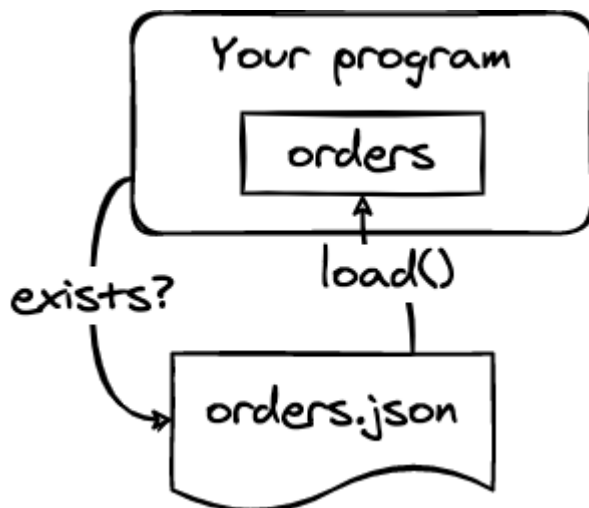
This chapter covers

- Erik and Simon create load and save functions in the main program
- Simon adds the exit function to the main menu and `get_order()` function
- Simon explains why he thinks Erik just created a real program
- The brothers discuss future plans

"Now let's get real," Simon said. "Yesterday we played with sample orders and simple programs. Today it's time to use what we learned in our real program."

"Yes, let's do it!" Erik said.

"Open your file `main_menu.py` where we wrote the main menu. We should add a couple of functions to work with the JSON file. Let's recall what we have to do first," and Simon pulled his diagram from yesterday.



"So we have to write a function that will load the list of orders from a JSON file. But first it has to check if the file exists. If it doesn't, we create an empty list and return it from this function. If it exists, we read from it, convert JSON to a Python list and return that list."

"Let me help you," Simon felt that it's a bit confusing for Erik. "Usually, in functions that work with files we pass the file name as an argument," and Simon started the function at the bottom of the file, right before the last line with `main_menu()`:

```
def load_orders(filename):
```

"Now we have to check if the file exists. There is a special function for that and we can find it in the `os` module."

"What is 'os'?" Erik asked.

"OS stands for 'operating system.' Operating system in the computer manages all files and programs. It works with your screen, your keyboard, your speakers, and video camera. On a typical computer, the OS can be Windows, macOS, or Linux. In our case we are going to ask the operating system if a file with such a name exists on this computer," and Simon added a line:

```
def load_orders(filename):
    if os.path.exists(filename):
```

"Look, we used the `os` module here. That means we have to import it the same way we imported the `json` module. In this program we haven't imported it yet, so let's import them both."

Simon moved the cursor to the very beginning of the file and added two lines:

```
import os
import json
```

He returned the cursor to the `load_orders()` function and continued his explanation. "If the file exists, we open it for reading, use the `json.load()` function to read from the file to the list `orders` and return the list." He added three lines to the function.

```
def load_orders(filename):
    if os.path.exists(filename):
        f = open(filename, "r")
        orders = json.load(f)
        return orders
```

"What if it doesn't exist? We just create an empty list and return it."

```
def load_orders(filename):
    if os.path.exists(filename):
        f = open(filename, "r")
        orders = json.load(f)
        return orders
    else:
        orders = []
        return orders
```

"Now load function is ready!" he said and looked at Erik.

"I don't think I could write it myself," Erik said.

"Of course, it looks complicated when you do it first time. But look, you can read it as if it was plain English, can't you?"

Erik looked at the function again and tried to read it. "If the file with 'filename' exists, then open the file. Save it in the object called 'f'. Then load from that 'f' file into 'orders'. Hmmm... yes, I can read it." He was surprised. He could read Python now and understand it!

"The next function is easier," Simon continued. "I think you can write it yourself if you look at our file where we practiced with sample orders. Look, right after the two `append()` operations there are three lines that we need here. We already have a function called `save_order()` that does nothing except printing 'Saving order...'. Let's replace it with a real one. I think it should be called `save_orders()`—plural, because now we know how to save a list of orders in a JSON file, right?"

Simon wrote the beginning of the function:

```
def save_orders(orders, filename):
```

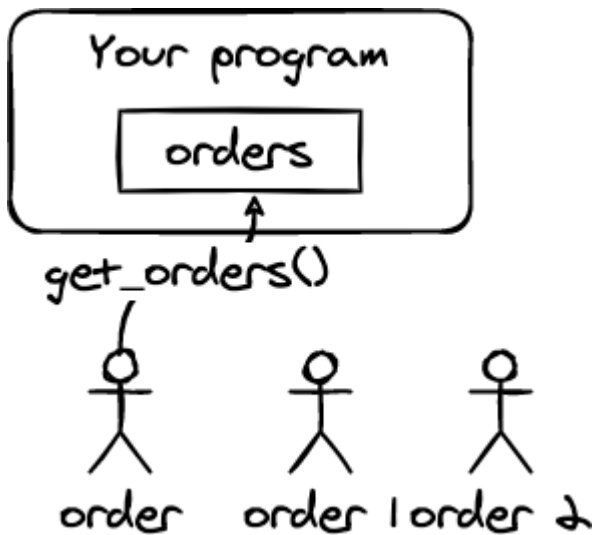
He explained: "We pass the list of orders as a first argument. Then we pass the name of the file where we want to store it. Now you can add those three lines from our yesterday's program."

Erik looked at the 'dict_json.py' file and copied three lines from it. Now the function looked like this:

```
def save_orders(orders, filename):
    f = open(filename, "w")
    json.dump(orders, f, indent=4)
    return
```

"Can we test it now?" he asked Simon.

"We are almost ready," Simon answered. "Look at the bottom of our file. Now we just call the `main_menu()` function. Remember my second diagram from yesterday?" and he pulled the drawing.



"This is what we do in the `main_menu()` function. We just have to edit it a little bit to serve several customers and save their orders in the list 'orders.' To do that we have to pass that list into the `main_menu()` function."

He moved the cursor to the beginning of the file and added the `orders` argument to the `main_menu()` definition.

```
def main_menu(orders):
```

"Now," he continued, "each time the customer enters a new order it will be added to the 'orders' list. Before we added the list as an argument, `main_menu()` didn't know where to add the new order. Now we can use the `append()` method and add it to 'orders.' Right after the customer confirms the order, we add it to the list of orders. We won't use the `save_order()` function here. We'll save all orders when you close the program." And Simon changed the `main_menu()` function to this:

```
def main_menu(orders):
    while True:
        order = get_order()
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel: ")
        if confirm == "Y" or confirm == "y":
            orders.append(order)
            print("Thanks for your order:")
            print_order(order)
        else:
            continue
```

"And we also change the main program to three steps: load the orders, main menu (get the orders), and save the orders." And he added those three lines to the bottom of the file. Now it looks like this:

```
orders = load_orders("orders.json")
main_menu(orders)
save_orders(orders, "orders.json")
```

"Can I try it now?" Erik asked.

"Sure, go ahead!" Simon said.

Erik ran the program, entered his name at the first prompt, then selected his drink components. When the program asked to confirm the order, he typed "Y." The program got back to the "What's your name: " prompt.

"Okay," Erik said, "it works. But how can I check my orders? Are they saved in the file?"

"Let's check," Simon said.

The brothers opened the 'orders.json' file and were surprised to see that it still contained the old orders from their yesterday's experiments. Even Simon was confused.

"Let's see," he said. "We open the file, we read from it, we get the order... But we never write to the file because we are still in the main menu! And we never reach that `save_orders()` function! Hmmm... let me think how to fix it."

Erik smiled. His know-it-all brother didn't know what to do.

That lasted only a moment.

Simon said: "I see. We didn't give the user a way to exit the main menu. We keep asking the user their name, but what if we want to finish the program?"

"I saw that you pressed `CONTROL-C` when you wanted to stop my program," Erik said.

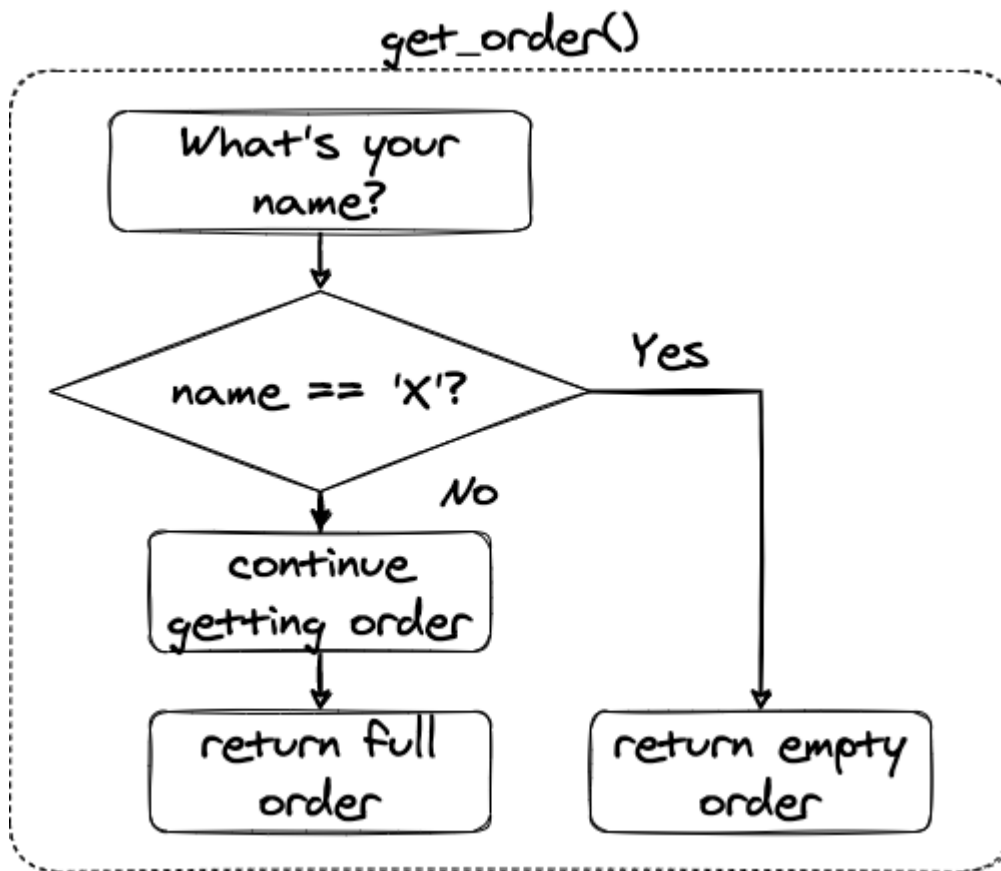
"Yes, I did it, but that's a not *normal* way of finishing programs. When I did that, the program was *interrupted*. Usually Python gives you an error message when you do that. When a program is interrupted that means it doesn't do anything else: it doesn't write our orders to the file, it doesn't close the file. Pressing `CONTROL-C` is a bad way to finish a program."

Simon paused for bit and continued: "We should give the user a *normal* way of finishing our program."

"Like Exit in the menu in Word?"

"Yes, like that. Let's tell the user that if they want to exit they should enter 'X' and only 'X' when asked about their name. The probability that we'll have a customer with a real full name 'X' is very low. Almost zero, actually. So let's do this: if in the `get_order()` function the customer enters name 'X,' then we don't ask any other questions and return an empty order, like this:

`order = {}`. Then this order goes to the `main_menu()` function and it decides: if the order is empty, it will save the order into the file and exit. If it's not empty, it will add the order to the list and continue working. Let's draw a diagram."



"This will be our updated `get_order()` function. Let me help you write it," and Simon took Erik's keyboard and started editing the `get_order()` function.

```

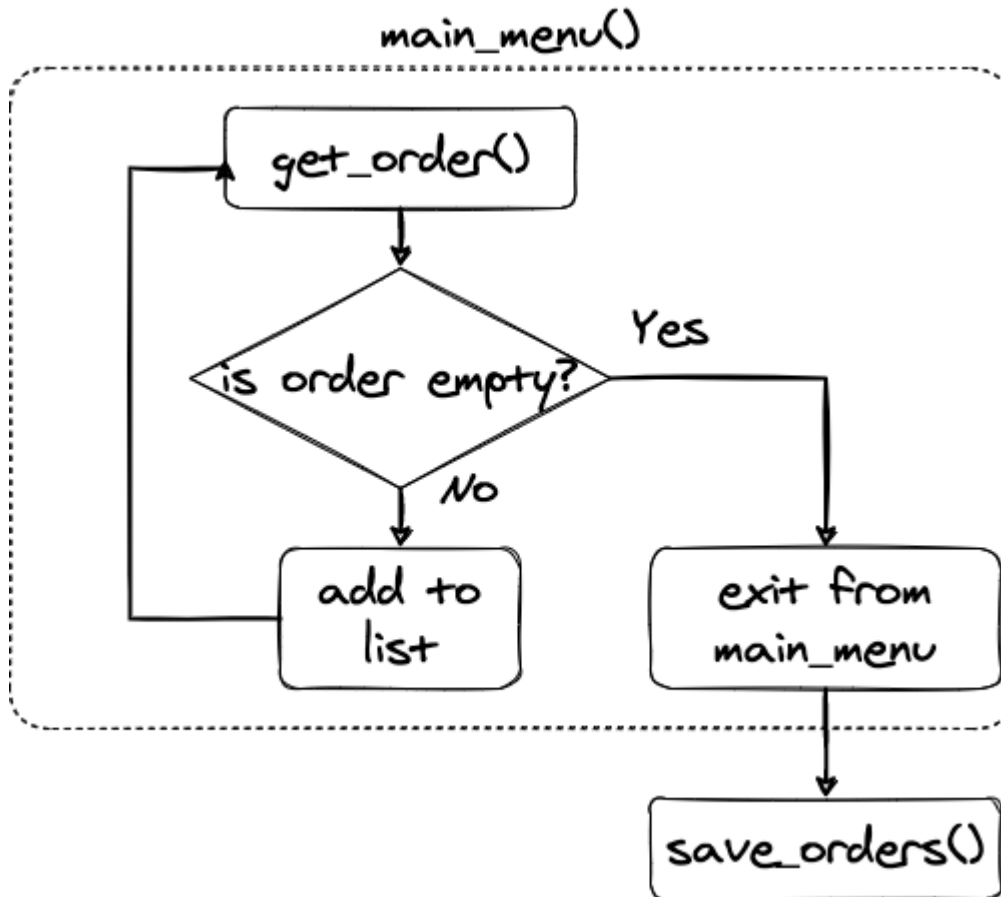
def get_order():
    order = {}
    name = input("Enter your name or enter 'X' to exit: ")
    if name == "X" or name == "x":
        return {}
    else:
        order["name"] = name
        drinks = read_menu("drinks.txt")
        flavors = read_menu("flavors.txt")
        toppings = read_menu("toppings.txt")
        order["drink"] = menu(drinks, "Erik's drinks", "Choose your drink: ")
        order["flavor"] = menu(flavors, "Erik's flavors", "Choose your flavor: ")
        order["topping"] = menu(toppings, "Erik's toppings", "Choose your topping: ")
    return order
  
```

"Why did you check both 'X' and 'x'? You told the user to type 'X' so they should type capital 'X', right?" Erik asked.

"Users usually don't bother pressing the Shift key. They may enter either 'X' or 'x'. So we should check both. The rest of the function should be clear to you—I just followed the diagram we

created together."

"Now for the main menu," Simon continued. "Here is another diagram," and he started drawing.



"As we discussed, if `get_order()` returns an empty order, we exit from the main menu. After that our program saves the orders in the file." Simon edited the `main_menu()` function to the following:

```
def main_menu(orders):
    while True:
        order = get_order()
        if order == {}:
            print("You entered 'X', exiting...")
            return
        print("Check your order:")
        print_order(order)
        confirm = input("Confirm? Press Y to confirm, N to cancel, X to finish: ")
        if confirm == "Y" or confirm == "y":
            orders.append(order)
            print("Thanks for your order:")
            print_order(order)
        elif confirm == "X" or confirm == "x":
            return
        else:
            continue
```

YOUR TURN Edit the main menu function

Edit the main menu function similar to what Simon did. If you need help, the full program text for this chapter is [here](#)

"And this is it," Simon said. "Let's test it. Just enter a different name this time so you will see that it was added to the JSON file."

Erik started the program. He answered "Jason" when the program asked about his name. He entered the rest of his order and typed "Y" to confirm the order. The program asked for his name again.

"Now let's enter 'x' and see if it exits properly," Simon suggested.

Erik typed 'x' and pressed `ENTER`.

You entered 'X', exiting... the program said and returned to the familiar `>>>` Python prompt.

"Now check the `orders.json` file," Simon said.

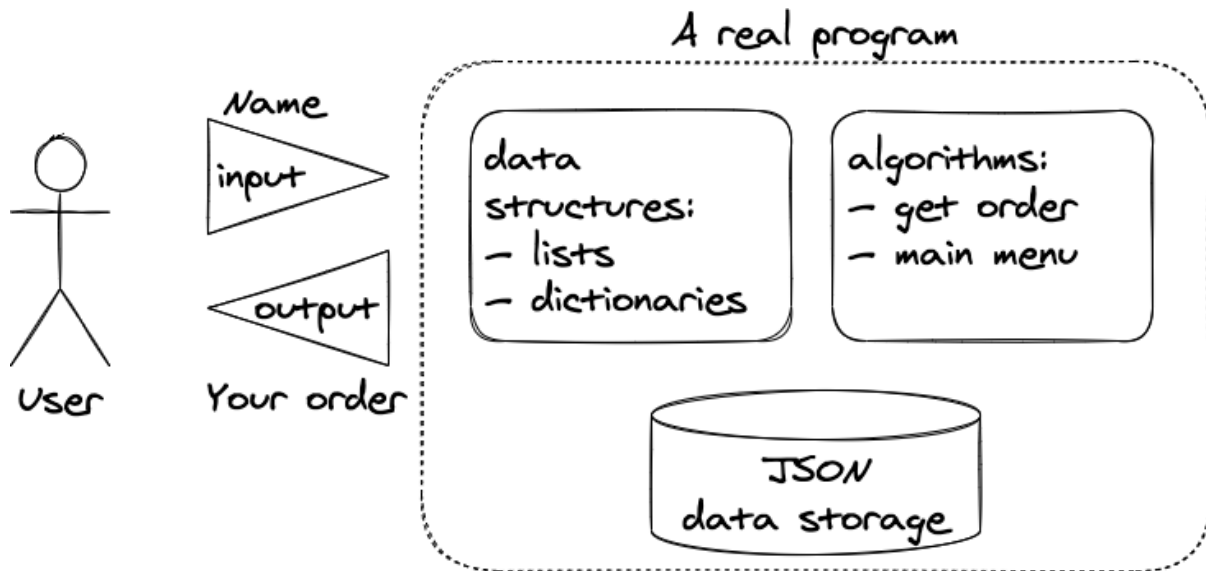
Erik started *TextEdit* and opened the JSON file. Right at the bottom of the file he found his recent order from 'Jason'.

"It worked!" he said. "It saved all the orders in the file and now I can see them all!"

"Yes, you can," Simon said and smiled. He was happy to see a complete working program that took orders, stored them in the file, and was written by his little brother!

"Congratulations, Erik!" Simon said. "I think now you can say that you created a real application. Look, it has input and output. It has data structures and algorithms. It checks for errors. It has data storage. And most importantly: it works and it's very useful—it collects orders. I am absolutely serious: it a good program and I am very proud of you."

As usual, Simon drew a diagram of what he called a *real* program.



"Yes, I like my program too," Erik said. "It does what I want, and it looks good. It prints orders almost the same way I saw it in Starbucks. Yes, almost... Maybe I can add a couple of lines or stars to make it better. I have some other ideas about what to add to this program."

"What else do you want to add?" Simon asked.

"First of all, I want to make it a web application. You know, with menus and buttons. It should be online so I could take my iPad with me and use it."

"Great idea!" Simon said. "Let's start working on it next week. I have a couple of ideas too," and he smiled.

"Why are you smiling?" Erik asked.

"I remembered how you thought you were done with the program after our first day."

"Ha, yes, I remember that too," Erik said. "Of course, the program was not quite ready then. What are your other ideas?"

Simon said: "I would add a couple of things to our data structure. For example, we can add the date and time when the order was made. That way we'll be able to see how many customers we served each day or each month."

"Yes, I think that will be good," Erik agreed.

"Then, maybe we should save orders in the data storage right after they are entered. That will make sure we keep all the previous orders even if the program fails and crashes."

"But you said it will make it slower," Erik remembered.

"Just a tiny bit. But it's worth it—otherwise we risk losing all our orders. I am thinking about

using a database for that."

"Also," Simon continued, "we need functions like 'print all orders' and 'count how many portions of vanilla flavor we used'. If we want to make your program a real business application."

"Of course, I want it," Erik said. "But first I want to make it a web application and make it beautiful."

"Sure, we can start working on it next week."

NOTE

To the reader

In the following chapters of this book we'll continue Erik and Simon's journey and develop a web application with them.

If you want to look at the other improvements that Simon suggested, you'll find them on the book companion site <https://github.com/pavelanni/lets-talk-python-book>.

9.1 New things you have learned today

- *How to check if a file exists*
We used the `os` Python module for that and the method `os.path.exists()`. You pass the file name and it returns `True` or `False`.
- *Pressing `CONTROL-C` is not the right way to finish a program*
We use it when we want to stop a program that behaves abnormally. Good programs should always give you a way to finish it normally.
- *What a real program is*
We learned that real programs have input and output, data structures and algorithms, data storage and error checking.

9.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch09>

10

Learning Flask: Your first web application

This chapter covers

- Erik creates his first simple web application
- Simon explains how Flask works (and what it is)
- Emily and Erik work on a web form
- The first coffee shop web-based menu is ready!

"You said you wanted to create a web application?" Simon asked Erik the other day.

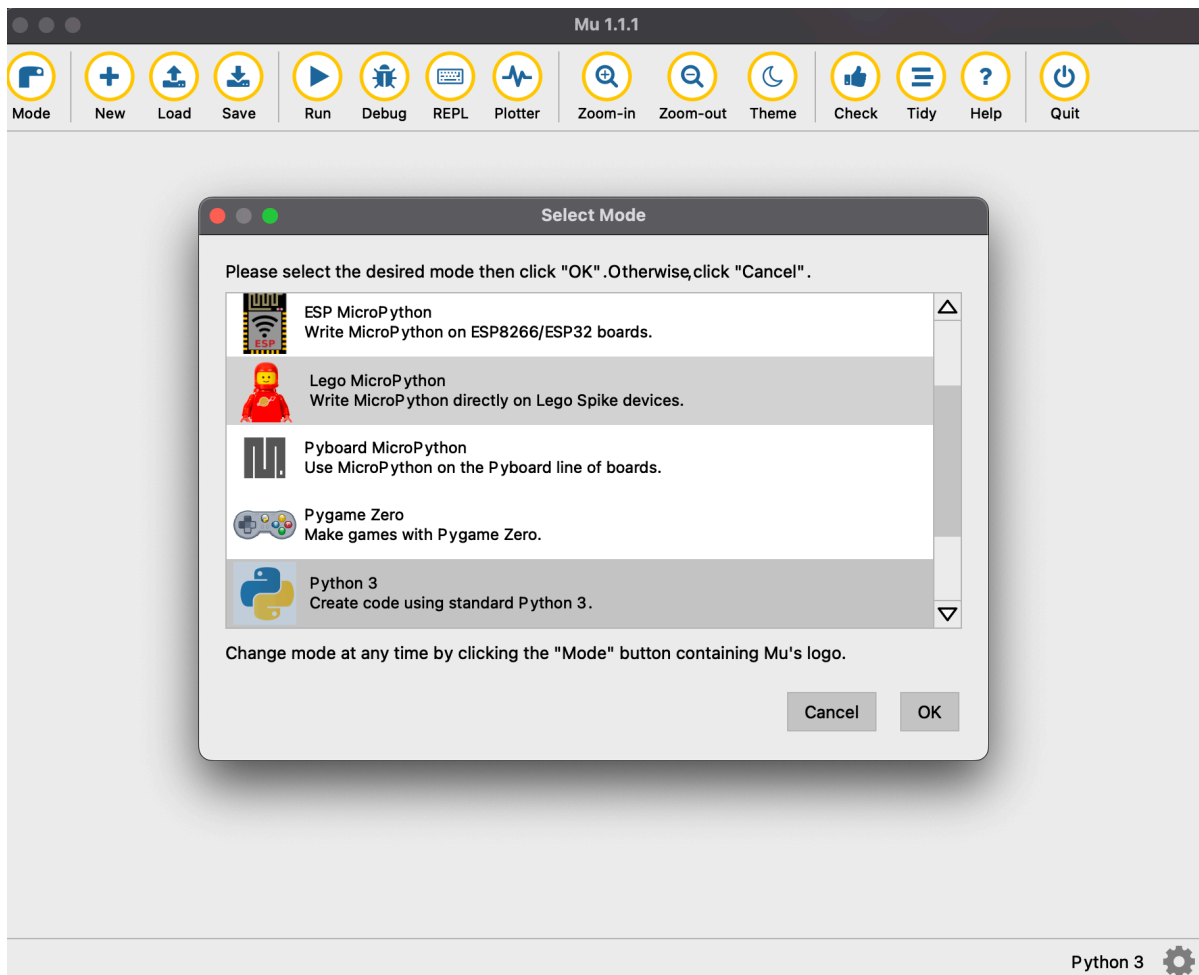
"Yes, sure!" Erik said. "Otherwise how could I use my program on my iPad?"

"Okay, but be prepared: it's not an easy task. It will require all your attention. Maybe you won't completely understand *everything* we do here. But not to worry, I'll help you when you need it."

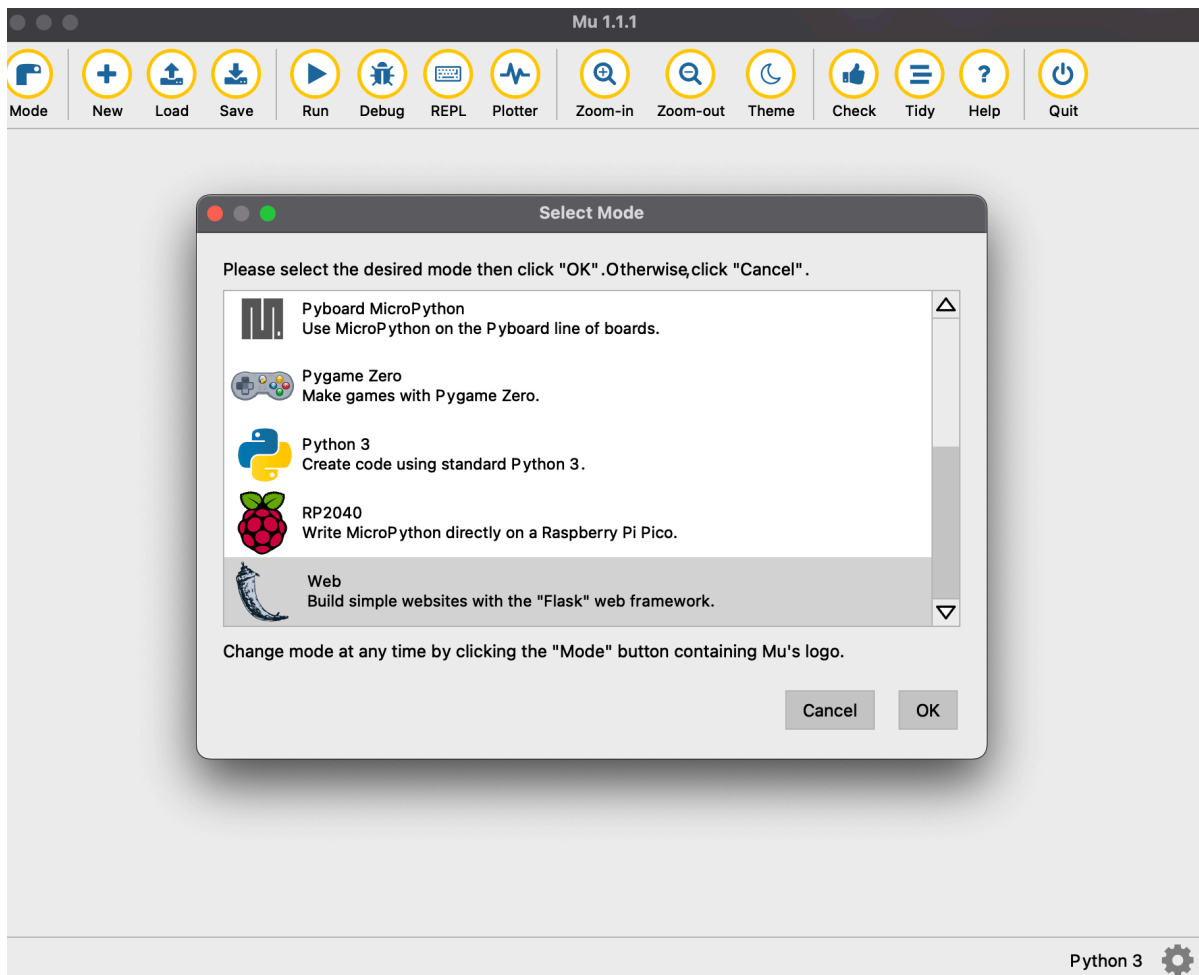
"I know, you are a good brother," Erik said and there was almost no irony in the way he said it.

"We will use our good friend Mu Editor for our web application. It has a special mode for that. Start the editor and click Mode in the top-left corner."

Erik did and saw this menu:

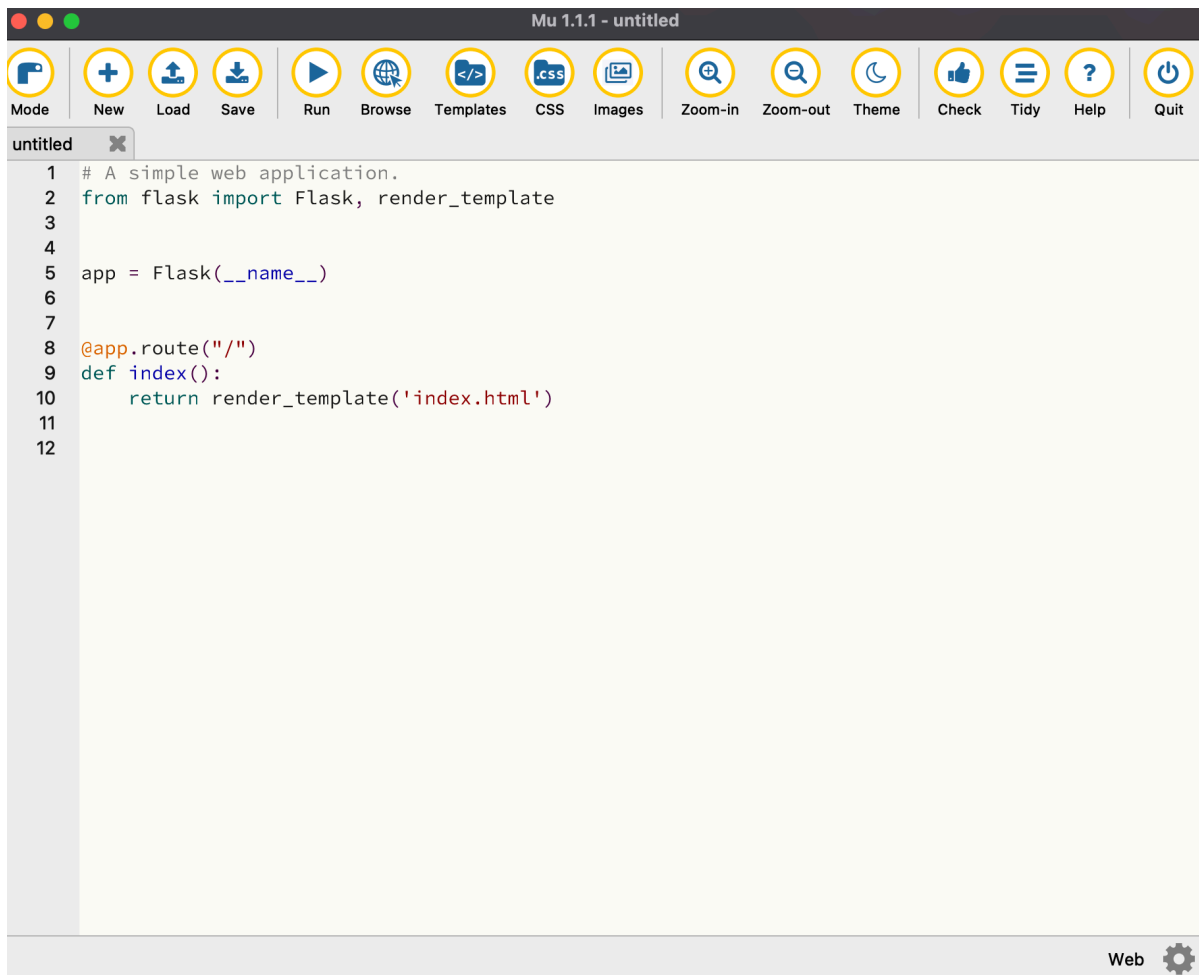


"Scroll to the bottom," Simon said. "Find the Web mode and click it. Then click Ok."



After Erik did that, Simon pointed to the bottom-right corner and said: "See this word 'Web' next to the cogwheel? We switched to the Web mode. Now let's see what we can do with it. Click New."

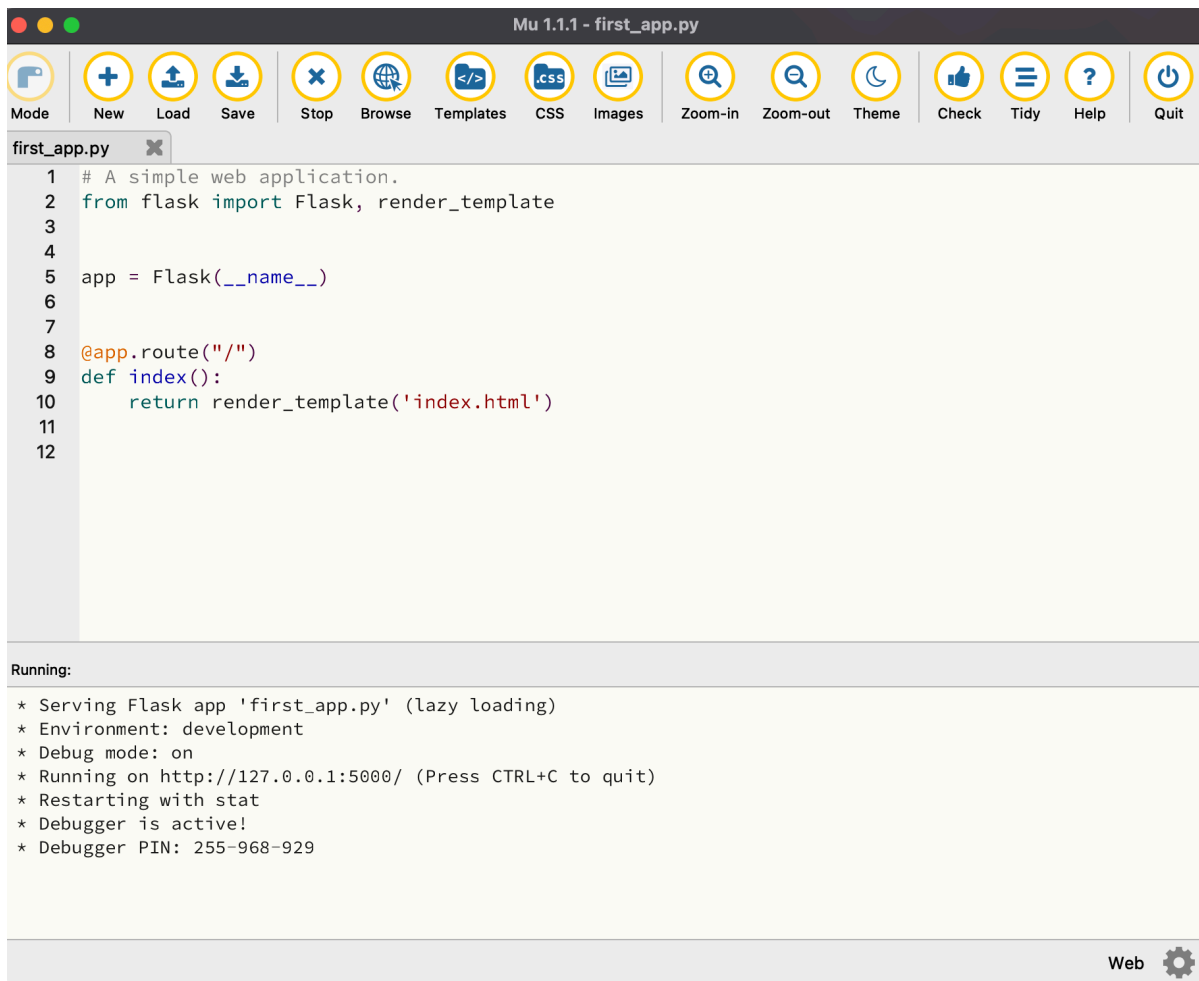
Erik did and immediately Python code appeared in the editor window.



"Interesting," he said. "Mu already wrote something for me. Can I run it?"

"Sure, go ahead. First you'll have to save it. Call it 'first_app.py'."

Erik clicked Run, entered 'first_app.py' in the Save dialog and saw this output at the bottom of the window.



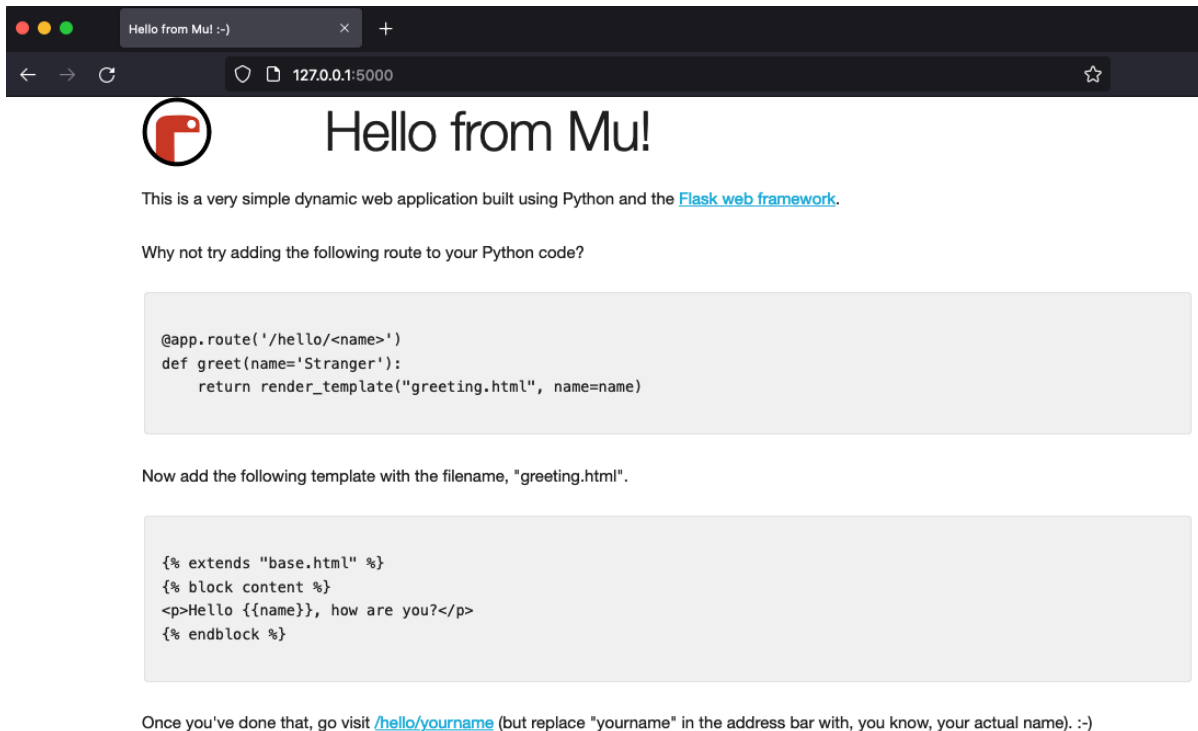
"What is it?" he asked.

"This tells you that your first web application is running. See this message: 'Running on <http://127.0.0.1:5000/>'. It means that you can go to your browser and enter this address: <http://127.0.0.1:5000/>. Sometimes this address is also called URL or Uniform Resource Locator—you will hear this word all the time when working with web. Or just click the Browse



Run button in the editor. Try and see what you've got."

Erik opened a new tab in his browser and entered the address. Here is what he saw:



"Wait, is this all written by my editor?" he asked Simon.

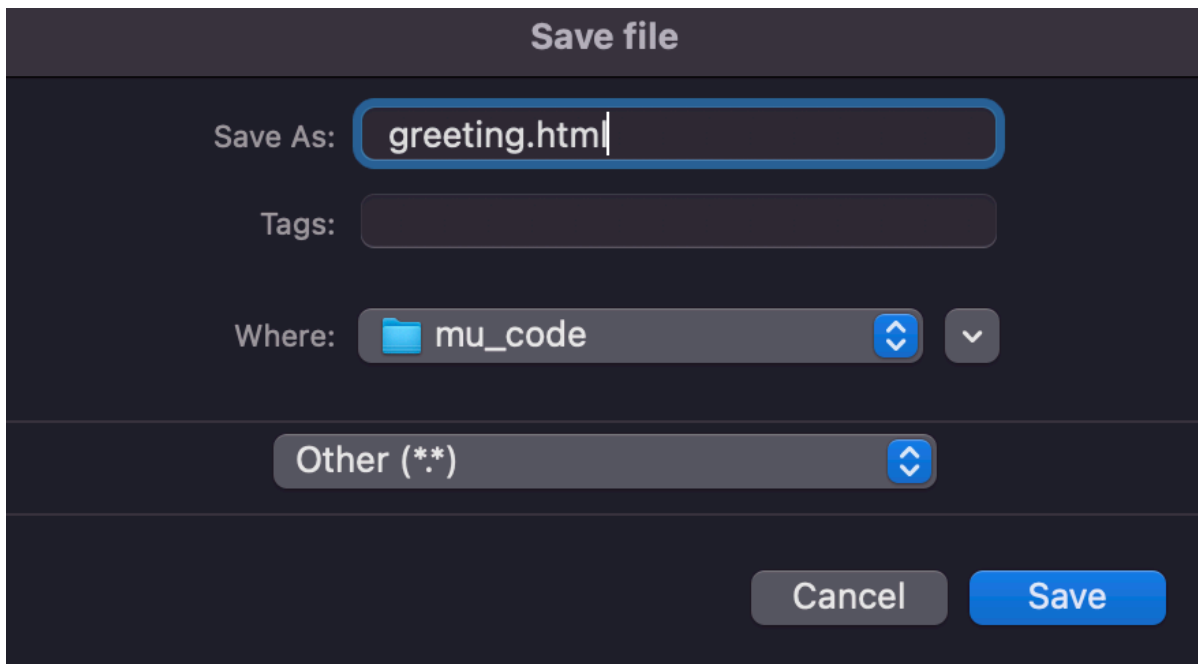
"Yes, but look, Mu suggests that the rest you write yourself," Simon said and pointed to the code example on the page. "Mu recommends you to copy the code from the first gray window to your program. Go ahead and do it."

That was easy. Erik quickly copied the text and pasted it below the existing code.

"Now Mu tells you to create a new file," Simon continued, "copy the text from the second window, and save it as a new file 'greeting.html'"

Erik knew how to do it. He clicked New in the Mu Editor, removed the program Mu put into it, and copy-pasted the text from the second gray window. Then he clicked Save.

Simon helped him: "Use the drop-down menu to change from '*.py' to 'Other (*.*)', otherwise Mu will think you are trying to save a Python program. We should tell it that this time it's a different type of file. In the 'Save As' field type `greeting.html`."

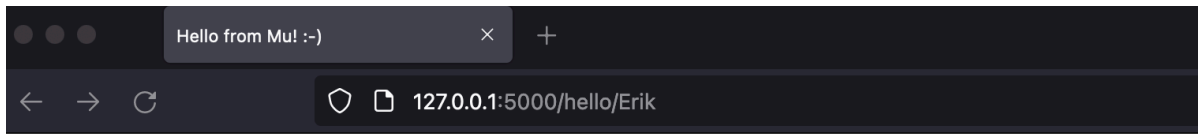


"Now stop the program and run it again," Simon said. "Don't forget to switch to the 'first_app.py' tab."

Erik switched to the application tab, clicked Stop and Run again. He saw the same output at the bottom of the window.

"Now go back to your browser and do what it suggests. Look, it says 'go visit /hello/yourname' and tells you to use your own name. Go ahead, add /hello/Erik in the address bar, right after 5000."

Now it looked like real hacking. Erik entered what Simon suggested and pressed `ENTER`.



Hello!

Hello Erik, how are you?

"Wow! It talks to me!" He was really impressed.

"It's already *your* program that talks to you," Simon said. "That was easy, huh?"

"Wait," Erik said, "if we are going to work on this web stuff, I should call Emily. She told me she learned HTML and this is what we need for web, right?"

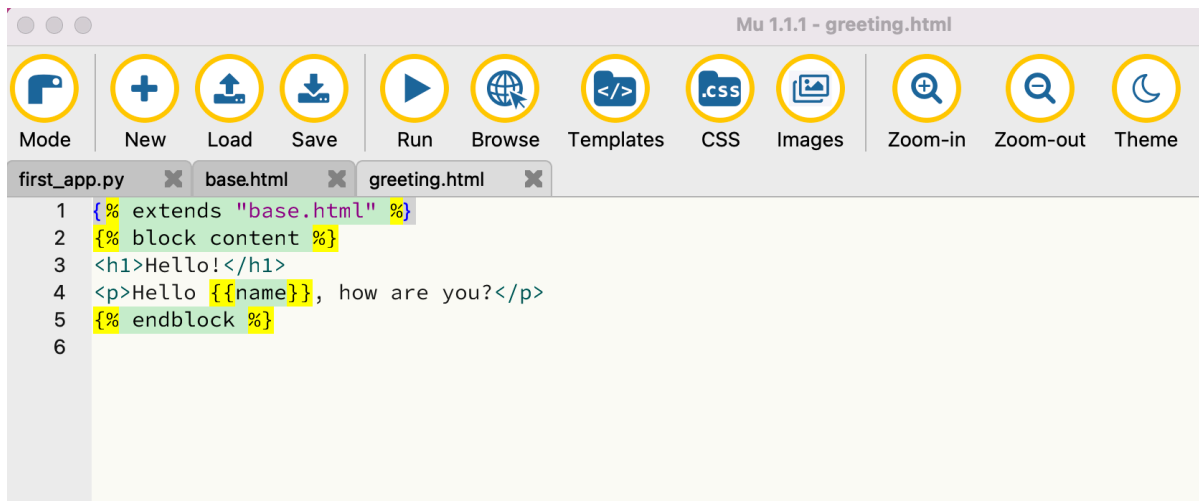
"Absolutely correct," Simon agreed. "Go ahead, call her. It's always good to work together."

YOUR TURN Create your first web application

Switch to the web mode in Mu Editor and create your first web application by copying the example from the browser's page, just like Erik did. Try to run it with your name. Try other names. Show it to your friends and ask them to use their names.

Emily lived nearby and she arrived in about 15 minutes, very excited about the project. She asked immediately: "Erik, show me your HTML!"

Erik showed her the `greeting.html` file and said: "Well, it's not *my* code, it's from this Mu Editor."



"Wow, interesting," Emily said. "I've never seen these curly braces in HTML."

"Right," Simon said, "because this is not pure HTML, it's a *template*. We use the program called Flask here and it uses templates to generate HTML."

"I see," Emily said. "But I know these `<h1>` tags, and `<p>` tags."

"Tags? What are 'tags'?" Erik asked.

"Tags are these small pieces of code that you put in your text to change how it looks. Look here, you place `<h1>` before 'Hello!' and `</h1>` after it and it looks larger. This is what in HTML is called *headers*, like chapter headers."

"What about `<p>`?" Erik asked.

"It means 'paragraph'," Emily explained. "In HTML you can write your text how you want: in one long line, or in many short lines, or even one word per line. But if it has `<p>` at the beginning and `</p>` at the end, it will be one paragraph in the browser."

"There are a lot of other tags," she continued. "You can make your text bold or italic, change colors, and all that."

"Emily, do you know anything about HTML forms?" Simon asked.

"They told us in the class that we can create forms in HTML to enter text or use menus," Emily answered. "But I haven't tried them myself."

"Menus is what we want!" Erik exclaimed.

"I'll help you," Simon said. "First we should use the mandatory HTML tags. We should always have `<html>` at the beginning of the file and `</html>` at the end. Also we should use the `<body>` tags around our text. Again, we use `<body>` to *open* the text and `</body>` to close it. That's why

the tags with slash / are called *closing* tags."

Listing 10.1 templates/forms.html

```
<html>
<body>

</body>
</html>
```

"They are like brackets in a list in Python," Erik said. He wanted to show Emily that he knew Python already.

"You are right," Simon confirmed. "These tags *enclose* some text and explain its meaning. Some pieces of text are headers, some are paragraphs. But now we want to create a menu. For that we'll need a tag `<form>` first and then a tag `<select>` inside it. Let's create a very simple menu," and he started writing.

Listing 10.2 templates/forms.html

```
<html>
<body>
<form>
  <select>
    <option>Coffee</option>
    <option>Decaf</option>
  </select>
  <input type="submit" value="Submit">
</form>
</body>
</html>
```

YOUR TURN Create your first web form

Create the file `forms.html` and save it under `mu_code/templates`. Copy the preceding code and test it in your browser. Try to change the options; try to add more options.

Simon finished writing, clicked Save, and saved the file as `forms.html` under the `mu_code/templates` directory. "Look, how many elements enclosed in tags do you see here? Emily, you should be more familiar with that."

Emily started counting. "First, the `<html>` tag, then the `<body>` tag. Inside the body we have a `<form>`, then inside the form we have `<select>`. It's for the menu, correct? And then in the 'select' we have two `<option>` elements."

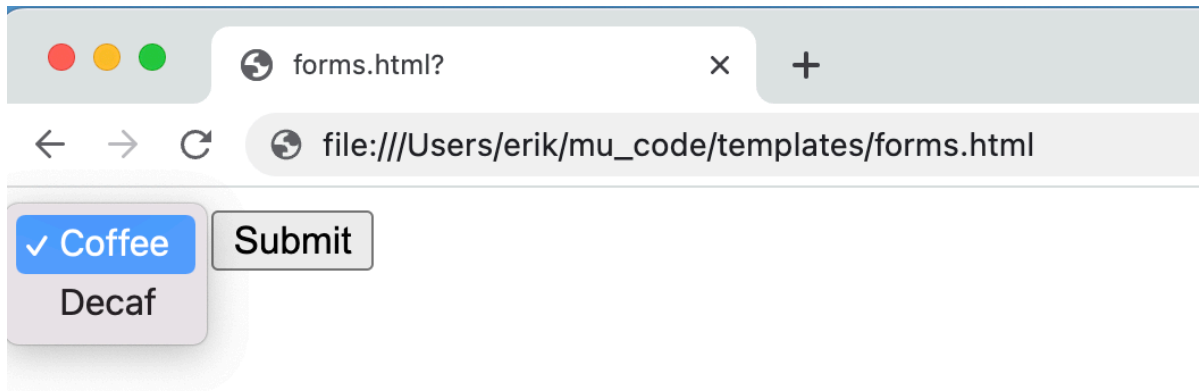
"Right. You did a great job, Emily," Simon said. "And also don't forget the `<input>` element that is a part of the 'form.' It doesn't have a closing tag. It exists just by itself. We use it to create the 'Submit' button."

"Let's see how it looks like in the browser," Simon continued.

"Can you open *files* in the browser?" Erik asked. "I thought browsers are only for web sites."

"Of course, you can," Emily answered. "We did it all the time in our HTML class! You just use the menu File in your browser, then Open File..., then find your file, and that's it."

Erik did what Emily just said and opened the file `forms.html`. He saw a menu, very similar to what he saw and used on many sites. He clicked the menu and it opened:



"I didn't know that you can create forms so easily," Emily said.

"Yes, it's pretty easy to create a simple form like this, but there are some missing parts," Simon said.

"It looks good to me," Erik said. "What's missing?"

"Yes, it *looks* good, but it doesn't *do* anything," Simon said. "We have to get data from the user and then *pass* that data to the program. How can we pass the data?" Simon asked, and answered his own question. "We should use variables and values, very similar to Python. Let me add something to this form."

Listing 10.3 templates/forms.html

```
<html>
<body>
<form action="/order" method="post">
  <select name="drink"> ❶
    <option value="">- Choose drink -</option> ❸
    <option value="coffee">Coffee</option> ❷
    <option value="decaf">Decaf</option>
  </select>
  <input type="submit" value="Submit">
</form>
</body>
</html>
```

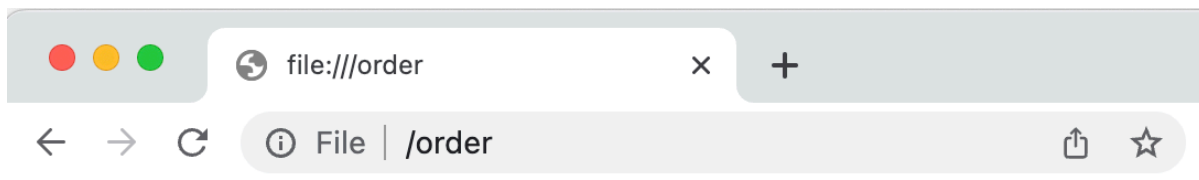
Simon started his explanation.

1. "Look at the number (1)," Simon said. "Here we define the *variable* we want to return from this menu. In this menu the variable is called `drink`."
2. "Now, on the line with number (2) and the line after it we define the *value* this menu option will return. It's very similar to what we did when we chose items from the text menu, remember?"
3. "And in the line with number (3) I just added an option that tells the user what to do. This option will show first in the menu and work as a prompt. As you can see, its value is empty. If the user haven't chosen a drink, we should tell them about it. You can't prepare their order without this information, can you?"

"Can I try it?" Erik asked.

"Of course, go ahead and open this file again. Or just reload it in the browser."

Erik reloaded the file, chose "Decaf" from the menu, clicked "Submit" and got this:



Your file couldn't be accessed

It may have been moved, edited, or deleted.

ERR_FILE_NOT_FOUND

"What's that?" he asked, visibly puzzled.

"Oh, I forgot to tell you," Simon said. "Look, I changed the `form` tag a little bit."

Listing 10.4 templates/forms.html

```
<form action="/order" method="post">
```

"Each form should have an *action*," he started to explain. "Action is something that our application will do when the user submits the form. When the user made their choice—coffee or decaf—they should *pass* this information to some function. That function should know what to do with this information: store it in the file or database, print the order, and such."

"Like what we did in our previous program?" Erik asked.

"You keep talking about your 'previous' program, can you show it to me?" Emily demanded.

"I'm sorry, Emily," Simon said. "I should have explained it earlier. We worked with Erik on a program that collects orders in a coffee shop. Similar to Starbucks, where you can order a drink, add flavors and toppings, and all that. Erik wrote a program that shows menus and asks the customer what they want to order. When they choose their drinks, flavors, and toppings, the program prints the order. But the program now works only in a terminal, in text mode. Erik needs your help to convert it to a web application."

"I see now," Emily said. "This sounds like a cool project! I hope Erik will teach me Python too."

"Of course," Simon said. "Teaching somebody is the best way to learn."

"Back to our form," he continued. "That `action` attribute tells the browser: 'After the user submits the form, open this address and pass the information from the form there. In our case the address is called `/order`. Don't worry, it only sounds scary, I'll show you what to do with it," Simon added because he noticed the confusion on Emily and Erik's faces.

"I still don't understand," Erik asked. "Where is this address that you are talking about?"

"Look at your first application," Simon said. "See this `greet` function?" "This function was written by the Mu Editor for us—or, rather, its authors," Simon said. "You see now the familiar function definition that's starting with `def`, but also, right above it, you can see something new: `@app.route('/hello/<name>')`. In Python it's called *decorator*, but we are not going to learn about decorators today.

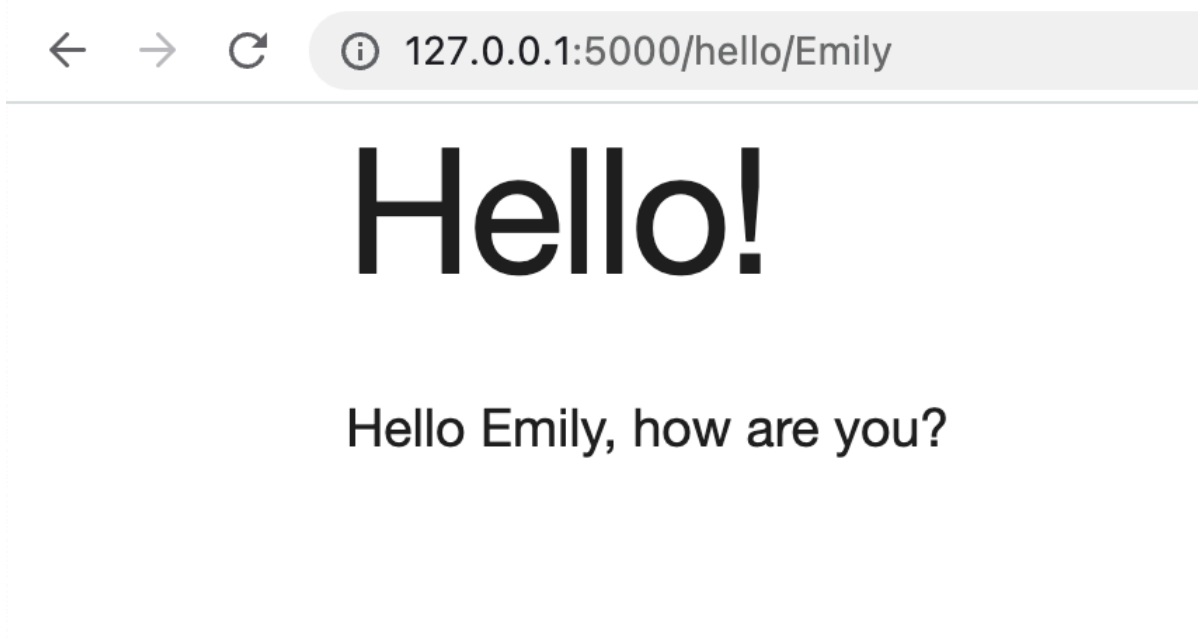
Listing 10.5 first_app.py

```
@app.route('/hello/<name>')
def greet(name='Stranger'):
    return render_template("greeting.html", name=name)
```

What's important for us today is that you can use it to tell your program which function to use for which address."

"Ah-ha, the address is that `/hello/Erik` that I entered in the browser, I see now!" Erik said. "Let's show Emily how it works," and he opened the tab with the `first_app.py` script in the editor, clicked Run and then Browse. His browser opened a page with greetings from Mu.

"Emily, look, I can type the address here, right after these numbers: `127.0.0.1:5000` and look what it shows us!" Erik typed: `/hello/Emily`, pressed `ENTER` and the browser showed:

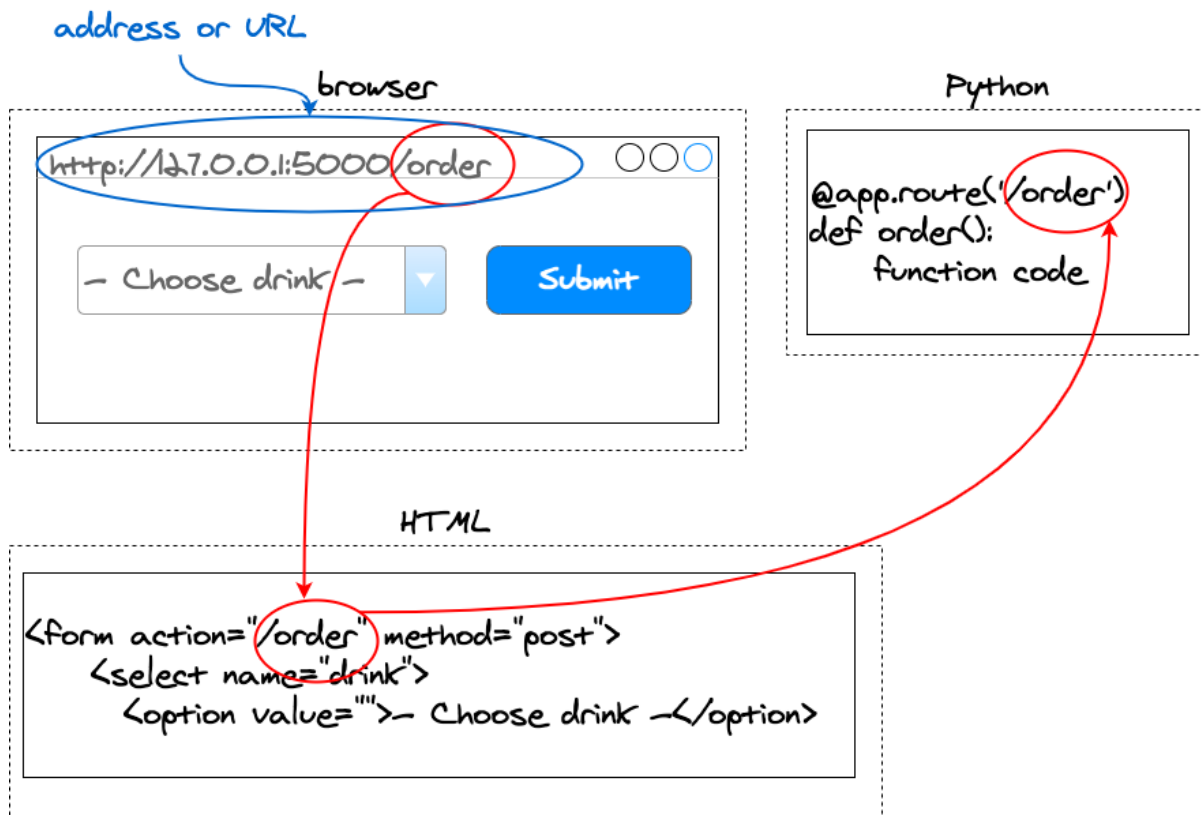


"Wow, I like it!" Emily said. "Can I try?" and she changed "Emily" to "Erik". Of course, the browser showed the page with "Hello Erik, how are you?"

"Interesting!" she said. "In our HTML class we could change pages, but we had to edit HTML. This is much easier!"

"Right," Simon said, "this is what we call *dynamic* pages—pages that change depending on what you enter. You can enter your information in the address, like `/hello/Emily` or you can use forms. Then the page will be *generated* using the information you entered. I'm sure, you've seen this kind of pages many times—for example when you enter a comment, or chat with someone on the web. You click 'Submit' or just press `ENTER` and the page is updated, right? Now you will learn how to make such pages yourself."

"Let me show it on a diagram," he said and started drawing.



"This combination of letters and numbers at the top of your browser is called 'address' or 'URL'. I marked it blue. Usually you see the site's name here, like `google.com`. In our case it uses numbers called the site's IP address. We use your own computer and for every computer in the world the address `127.0.0.1` means 'this computer.' But don't worry about that now."

Simon pointed to the first red circle around the word 'order' in the address and said: "This is what we should care about. Look, it's part of the address. When we open this address we see the form with the drinks menu. When you click Submit," and he followed the arrow down his diagram, "the form knows that it should find the function responsible for the address `/order`. You see, it's here, in the form `action` field."

And then," he followed the arrow up to the 'Python' block, "the form finds the Python function that can work with it, because we used this decorator, `@app.route('/order')`. You see, these three things are connected; you just have to use the same name in the address, in the form, and in the Python program."

"I see that the function is also called 'order'--is it the fourth place where we use it?" Erik asked.

"You have a very sharp eye!" Simon said and smiled. "No, in this case, the function can have a different name. I could call it 'new_order' or 'get_order'. But now we have to write the actual function. I'll help you here. It will look a bit scary, but don't worry. I'm learning this *Flask* system myself and usually I follow the online tutorial and take examples from there. Don't think that I remember all these things myself."

And Simon wrote the function, looking at the example he kept opened in the browser. He added numbers at the end of the lines to help him in his explanation.

"At the line with (1) you see two words: 'GET' and 'POST'. These are the *methods* that we use with web servers. We use GET when we want to get something from a web server, like a web page. We use POST to *send* some information to the web server. Like in this case—we want to send or POST the drink chosen by the customer. Put it another way: when you load a page in the browser—you use GET; when you click Submit in your form—you use POST. You will understand it better when we start using it, don't worry.

"And here, at the line with number (2), we start using one of the words. Look what it says: if the method is POST--which means somebody filled the form and clicked Submit—we read the information they entered in the form and print it.

"Now look at the number (3). Remember in the form we used `<select name="drink">`? This is the name we use here, in the square brackets. Later we'll add other menus—for flavors and toppings. In the form they will have names like 'flavor' and 'topping'. Here in the code we'll use them as `request.form['flavor']` and `request.form['topping']`."

"Under number (4) we just print whatever we received from the form. You will see it in the editor.

"In the line with number (5) we tell our web server to print this page with the menu. It's like a menu loop that we used in our program before—you get the information from the customer, print it out and return to the menu to get another order. And you repeat this loop until you are done entering orders."

Listing 10.6 first_app.py

```
@app.route('/order', methods=('GET', 'POST')) ❶
def order():
    if request.method == 'POST':              ❷
        drink = request.form['drink']          ❸
        print("Drink: ", drink)               ❹
    return render_template("forms.html")      ❺
```

- ❶ Methods that we are going to use with this form
- ❷ Method POST means we are submitting information
- ❸ We get the customer choice from the form's field called `drink`
- ❹ Print the choice we've received
- ❺ Display the template `forms.html`

"Let me add one more thing," Simon said and added `request` to the first line with `import`. "This

module called `request` is a part of Flask. If we use it we have to import it." Now the first line looked like this:

```
from flask import Flask, render_template, request
```

YOUR TURN Write your own `order()` function

Add the `order()` function from above to your `first_app.py` program. Don't forget to change the `import` line. Try to run it. Open a new tab in the browser and use the <http://127.0.0.1:5000/order> address. If you are having problems, continue reading and follow what Emily and Erik are doing.

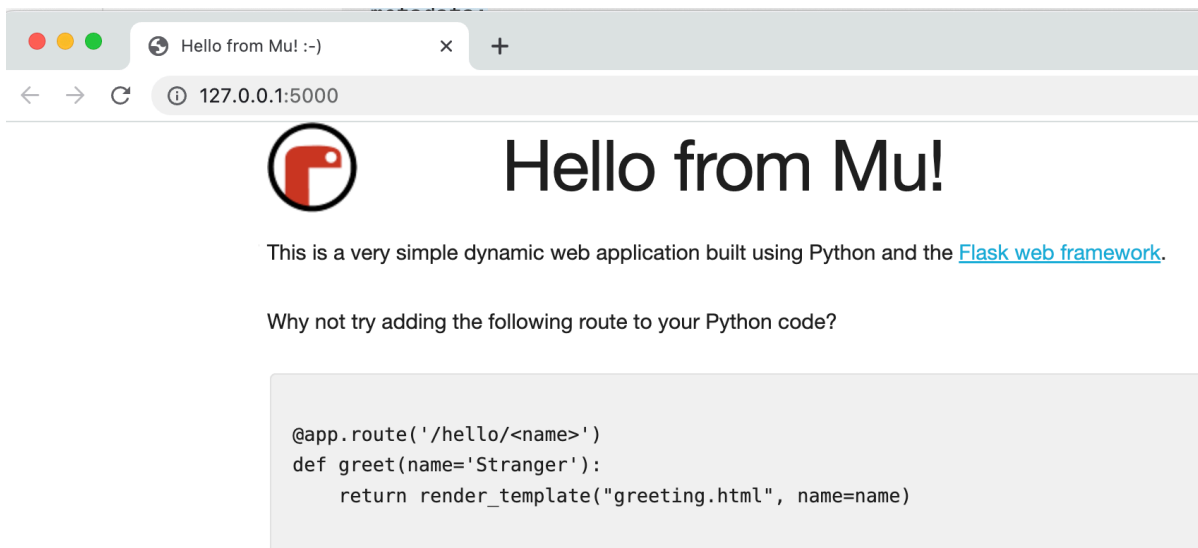
"Can I run it now?" Erik asked. He was a bit tired after such a long explanation. Emily, on the other hand, listened to Simon's explanations as if he was a wizard. She liked all this programming magic and couldn't wait to try the program.

"Can I run it?" Emily asked.

"Of course," Erik said. "Just click Save and then Run."

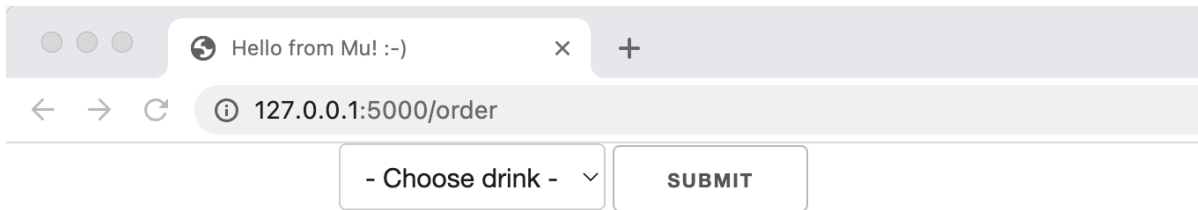
"Now click Browse," Simon said.

Emily did and the following page opened in a new browser tab.



"Now we have to add `/order` to the address, remember?" Simon helped her.

Emily did and the address line in the browser became this: <http://127.0.0.1:5000/order>. The page showed the menu they just created together.



"Go ahead, choose your drink and click Submit," Simon said.

Emily chose Decaf and clicked Submit. She's got the same page with the 'Choose drink' menu.

"Did it work?" she asked. She looked confused.

"Let's check," Simon said. "Go back to the editor. Look at the bottom window."

```
Running:
127.0.0.1 - - [16/Oct/2022 12:22:56] " [36mGET /static/img/logo.png HTTP/1.1 [0m" 304 -
127.0.0.1 - - [16/Oct/2022 12:27:22] "GET /order HTTP/1.1" 200 -
127.0.0.1 - - [16/Oct/2022 12:27:22] " [36mGET /static/css/normalize.css HTTP/1.1 [0m" 304 -
127.0.0.1 - - [16/Oct/2022 12:27:22] " [36mGET /static/css/skeleton.css HTTP/1.1 [0m" 304 -
Drink: decaf
127.0.0.1 - - [16/Oct/2022 12:30:23] "POST /order HTTP/1.1" 200 -
127.0.0.1 - - [16/Oct/2022 12:30:23] " [36mGET /static/css/normalize.css HTTP/1.1 [0m" 304 -
127.0.0.1 - - [16/Oct/2022 12:30:23] " [36mGET /static/css/skeleton.css HTTP/1.1 [0m" 304 -
```

"Do you see this line: `Drink: decaf`? It's what our program is printing," Simon said. "That means it works!"

"But I thought," Emily said, "that it would print the order on the page."

"It will, trust me," Simon said. "We haven't written that part yet. We are getting there, right now."

He took the keyboard and changed the `first_app.py` file by adding one line after the `print()` line. Simone explained: "When we first open the address `/order` in the browser that means we use the `GET` method. We want to *get* the page first, right? We don't have anything to `POST` yet. In that case we use the template `forms.html` that displays our drink menu. But after we have chosen a drink and clicked Submit we use the `POST` method. We want to *send* this information to

the program. And in that case we collect the data from the form—the drink choice—and use *another* template. I called it `print.html` because we want to print the order."

The `order()` function now looked like this:

```
@app.route('/order', methods=('GET', 'POST'))
def order():
    if request.method == 'POST':
        drink = request.form['drink']
        print("Drink: ", drink)
        return render_template("print.html", drink=drink)

    return render_template("forms.html")
```

"But we don't have a file called `print.html`," Emily said.

"Right, I am going to create it right now," and Simon created another file in the editor and saved it as `print.html` under `templates`.

Emily looked at his code and said: "Oh, this I can understand! You print the header 'Thanks for your order' and then you open a new paragraph and print 'Your drink' and then in **bold** you print the drink itself. And this `drink` in double curly braces works the same way it worked with my name when it printed 'Hello Emily', right?"

"Exactly right!" exclaimed Simon. "You are absolutely correct, Emily!"

Listing 10.7 templates/print.html

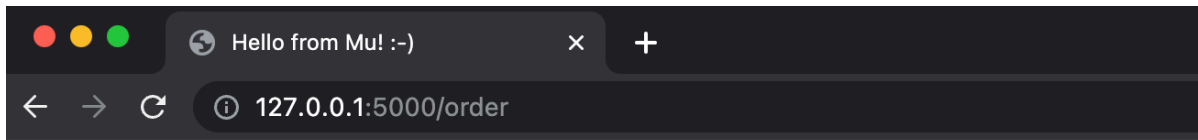
```
{% extends "base.html" %}
{% block content %}
<h1>Thanks for your order!</h1>
<p>Your drink: <strong>{{drink}}</strong></p>
{% endblock %}
```

YOUR TURN Edit your web form to print the drink choice on the page
Copy the template above into your `print.html` file. Feel free to change the title and the text. Change the `first_app.py` by adding the line with `return` and save it too. Try to run your program.

"Can I try it?" Emily asked.

"Sure, go ahead and click Run," Simon said.

Emily clicked Run, chose "Coffee" from the menu and saw this page:



Thanks for your order!

Your drink: **coffee**

"Yeah, it works!" she exclaimed.

"It's like calling the `print_order()` function in our previous program," Erik said.

"Yes, exactly!" Simon said.

"But how do I get back to the order page?" Erik asked.

"You see, the address in the browser is still pointing to `/order`. That means if you click the address line with mouse and press `ENTER` you'll reload the order page. Just don't click the reload button or it will create another order."

Emily did what Simon said and saw the order page again.

"But there is a better way," Simon said. "You were looking for a button on the print page, like Back to the order page, weren't you?"

"Yes, that would be easier," Emily agreed.

"We can use another form for that," Simon said. "It will be very simple," and he added several lines to the `templates/print.html` file.

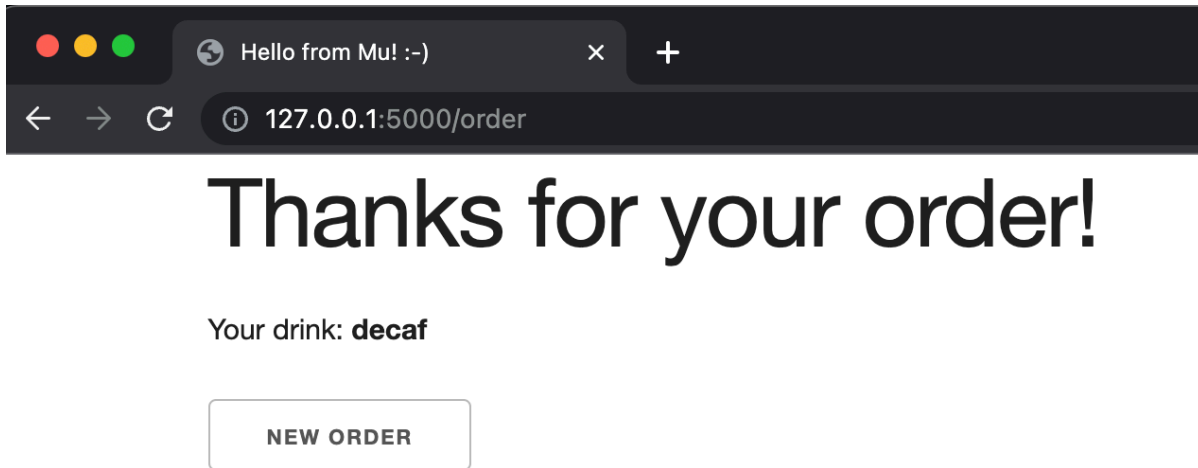
"Look, we created another form that has only the Submit button. We just renamed it to 'New order'. Look, it points to `/order` in its `action` field. That means when we click the 'New order' button it will send us to the `/order` page. And it will show the drink menu again. Try it!"

Listing 10.8 templates/print.html

```
{% extends "base.html" %}
{% block content %}
<h1>Thanks for your order!</h1>
<p>Your drink: <strong>{{drink}}</strong></p>

<form action="/order">
<input type="submit" value="New order" />
</form>
{% endblock %}
```

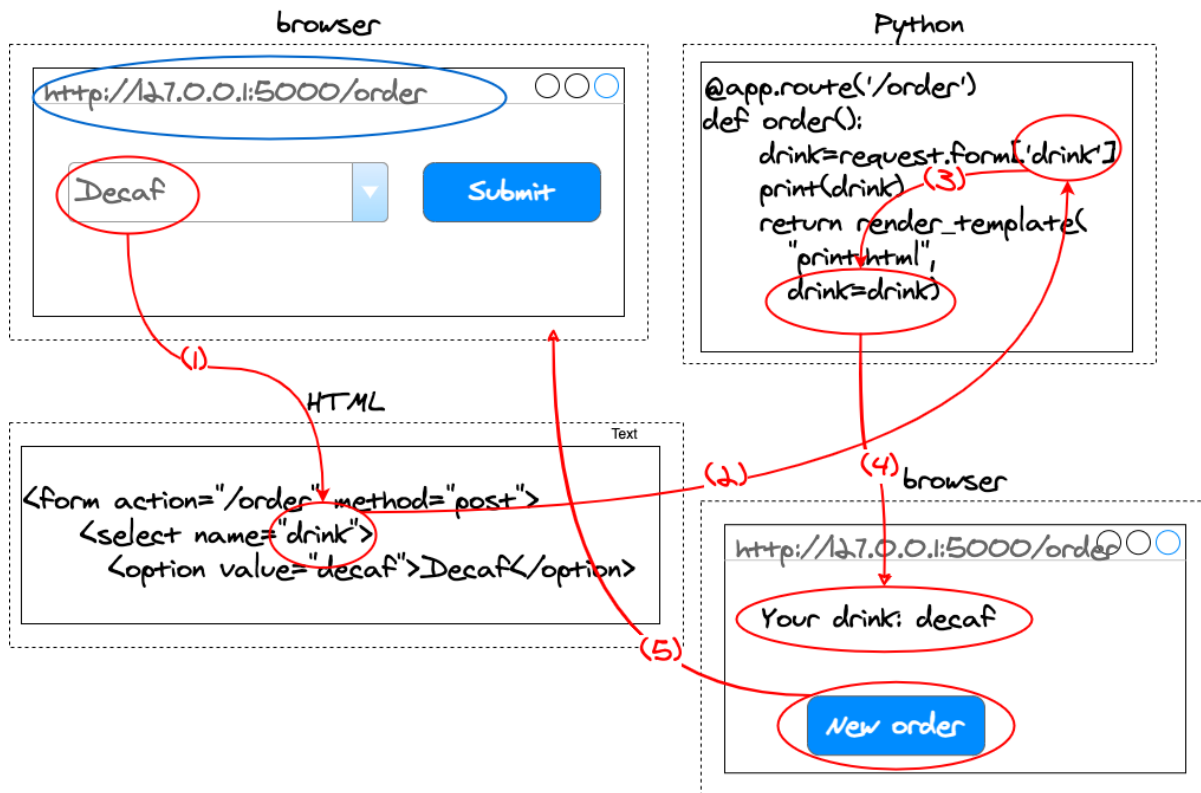
Emily did and after she clicked Submit she saw a page with the new button.



She clicked the button and got back to the order page.

Erik noted: "It's like our main menu with a loop. Order, then confirm, then print, then back to the order menu."

"You are right!" Simon confirmed. "Let me show on a diagram."



"Look at the diagram. Step one: you choose Decaf from the menu. This assigns the value `decaf` to the variable `drink` in the form."

"Step two: that value `decaf` is passed to our Python program via `request.form`. Now the variable `drink` in the Python program has the value `decaf`."

"Step three: we pass the value of variable `drink` from Python—which is `decaf`--to the variable `drink` in the template `print.html`."

"Step four: we call `render_template()` with the variable `drink` which is replaced with its value, `decaf`. And now `decaf` is printed on the web page."

"Finally, step five: we click 'New order' and return to the order page."

YOUR TURN Change the `print.html` template
 Add the 'New order' button to the `templates/print.html` template. Test it.
 Can you return to the order page?

"Emily and Erik, you did a great job today," Simon said. "Most importantly, you didn't fall asleep during all those long explanations."

"I almost did," Erik said.

"Yes, I noticed," Simon smiled. "But seriously, creating web applications is 10 times more difficult than working on text menus and dialogues. I admire your patience!"

"But it was worth it," Emily said. "The program works now!"

"We also need to add flavors and toppings," Erik said.

"Right!" Simon exclaimed. "We have to create all the menus but in the web form. I'm pretty sure Emily will help you with that."

"Sure," Emily said, "looks like we need to add more `select` forms to the template. Erik, will you show me your previous program? We can start working on the web application tomorrow."

"Of course," Erik said. "Let's get together tomorrow and work on that."

"I'll be happy to help," Simon said. "Please let me know when you start."

10.1 New things you have learned today

- **Web mode in Mu Editor**

In addition to the standard Python mode Mu editor also has a web mode. It has a simple web application example.

- **HTML Forms**

This is a way to get information from users into a web application. We can have menus, text fields, buttons. When we click Submit, the form sends information to a special address configured in the `action` field. From that address the information can be processed by a program.

- **Flask**

A program to help us create web applications. It is being developed by the Open Source community, it has good tutorials and examples. It is used by many online sites and web applications.

10.2 Code for this chapter

You can find the code for this chapter here:

<https://github.com/pavelanni/lets-talk-python-book/tree/main/ch10>

Ideas for your first application

Creating a coffee shop application doesn't look very attractive to you? Create something else! All programming ideas and methods we discuss in this book are applicable to a lot of other projects. Just look around and you'll get ideas for other applications.

For example:

A.1 Pizza place

This should be very similar to what we do with the coffee shop application. Look—we ask the customer:

- What is their main drink?
- What is the flavor they want?
- What is the topping?

We give the customer a list of options for each question and the customer chooses from the menu.

What should we ask the customer in a pizza place?

- What kind of crust do you want? Thin or thick?
- What size? Small, medium, or large?
- Which sauce do you want? Red or white?
- What kind of pizza do you want? Margherita, pepperoni, veggie,... Go to your favorite pizza place and see what they have.
- What additional topping do you want to add?

A.2 Ice cream shop

Go to your favorite ice cream shop and watch how they prepare your order. What do they ask you? What are the options they give you? Those will be in your application's menus.

Most likely they ask:

- What type of cone? Sugar, waffle, cake?
- How many scoops?
- Which flavors?
- Any topping?

Here it's becoming slightly different from what we did for the coffee shop. After you asked "*How many scoops?*" you have to ask *that many times* about the flavor. Think about it: how would you do it in Python?

Hint: there is a function `range()` in Python that can be used in a `for` loop. We used it in our menus. Try to use it to ask about the ice cream flavor the exact number of times.

A.3 LEGO® minifigures

You have a good collection of LEGO minifigures and their parts. You want to help your friends to build something new. What questions are you going to ask them and what options will you give them?

- Choose the head: smiley face, sunglasses face, face with a beard,...
- Choose the headwear: dark hair, blond hair, hard hat, police hat,...
- Choose the torso: mechanic, police officer, shirt with tie, t-shirt,...
- Choose the legs: blue jeans, green shorts, brown cargo pants,...
- Choose the accessory: a sword, a radio, a hammer, a magnifying glass,...

You can add special conditions to your application. For example if your friend have chosen a police hat then they can't choose a baseball bat as an accessory. Think about adding this condition to your menus.

What about choosing parts at random? That might create some funny minifigures. How would you add a random option to your menu? How would you implement it?

Hint: there is a module called `random` in Python. You should import it with the `import` statement in the beginning of your script and use the function `choice()`. That function works like this: you give it a list of choices and it chooses randomly one of them. Next time you call it the function randomly chooses something else (or, maybe, the same item—it's random!). For example, create this short script and run it. In this script we ask Python 5 times to choose randomly an item from the list of three types of hair.

Listing A.1 choice.py

```
import random

for _ in range(5):
    print(random.choice(["dark hair", "blond hair", "red hair"]))
```

Run this script and you'll see something like this:

```
blond hair
red hair
blond hair
blond hair
dark hair
```

Of course, in your case the list will be different and will have 5 *other* random choices in different order.

A.4 Other project ideas

Do you have other project ideas? Please share them in the liveBook forum: <https://livebook.manning.com/book/lets-talk-python/discussion>.

How to install Mu Editor and Python environment

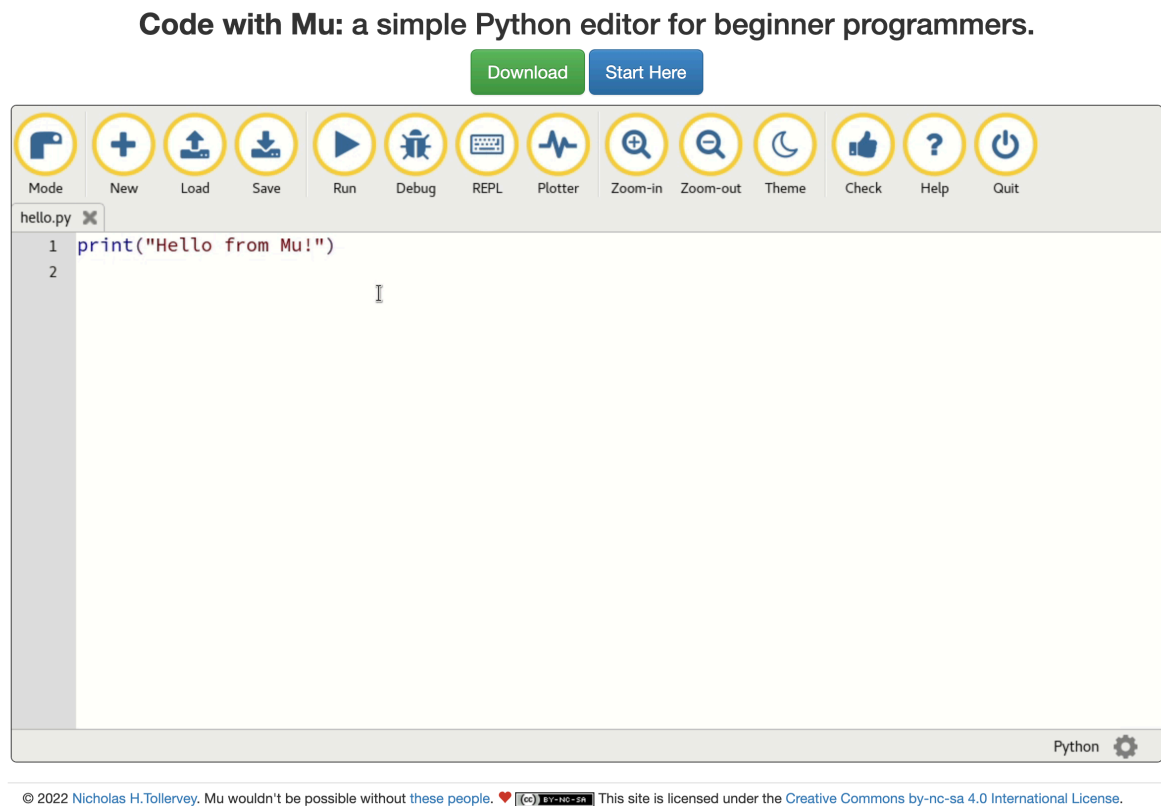


In this Appendix we'll explain how to install Python on your computer. The easiest way is to install a programming editor that contains Python in it. We recommend to install Mu Editor. We use it in this book so it will be easier to follow the book's dialogues and instructions.

We'll also give you links to other ways to install Python—feel free to try them too.

B.1 Mu Editor

1. In your web browser open the Mu Editor's web page: <https://codewith.mu/>.



2. Click Download (the green button), You will see the following page:


[Download](#)
[About](#)
[Tutorials](#)
[How to..?](#)
[Discuss](#)
[Developers](#)
[Language ▾](#)

Download Mu

The simplest and easiest way to get Mu is via the official installer for Windows or Mac OSX (we no longer support 32bit Windows). We also have an experimental AppImage for Linux users running on Intel based hardware.

The current recommended version is Mu 1.2.0. We advise people to update to this version via the links for each supported operating system. All previous beta versions of Mu [can be downloaded from here](#).



Windows Installer

[Download](#)
[Instructions](#)


Mac OSX Installer

[Download](#)
[Instructions](#)


Linux AppImage Package (Experimental)

[Download](#)
[Instructions](#)

3. Click Download for your operating system. Your browser will download the installation file for your operating system.
 - For Windows it will bedownload the installation file for your operating system.
 - For Windows it will be an `.msi` file.
 - For macOS it will be a `.dmg` file.
 - For Linux it will be an `.AppImage` file.
4. Click Instructions for your operating system and follow the instructions.
5. Open Mu Editor as you normally open applications in your operating system. You are ready to work on your project!

You can also use Mu Editor to program microcontrollers and build robots, but it's a topic for another book.

B.2 Thonny

Thonny is another great Python editor created with beginners in mind. You can find it here: <https://thonny.org/>

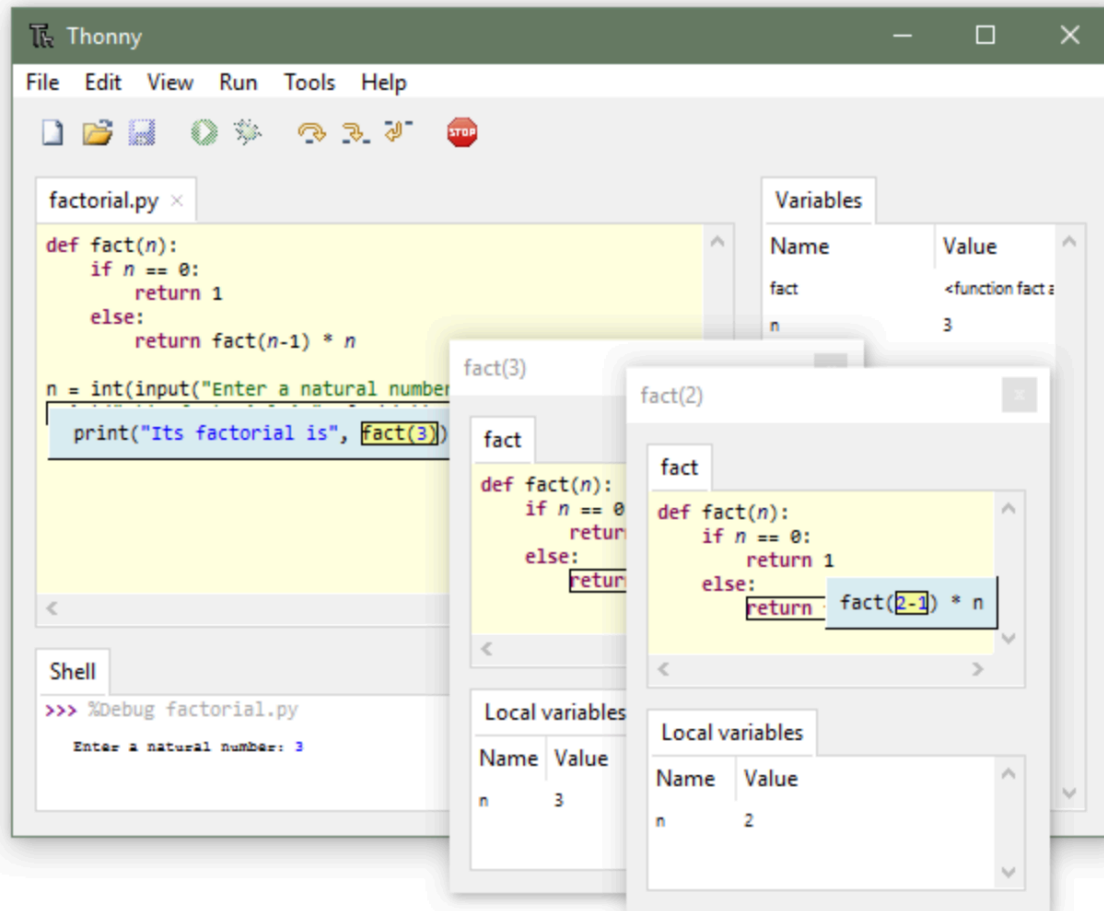
Right on the first page you'll find the installers for Windows, macOS, and Linux. For Windows download the `.exe` file and run it. For macOS download the `.pkg` file and install it. For Linux run the script provided in the instructions.

Thonny

Python IDE for beginners



Download version [4.0.1](#) for
Windows • Mac • Linux



After you installed the application start it and explore its settings. You can choose color theme from a dozen of options, editor and terminal fonts, and many other things.

Thonny has a very helpful feature called Assistant. In the Options menu you can configure it to start each time when there is a warning in your code. Also it starts when your program shows an error. Assistant gives you several suggestions on what could be wrong with your code. Try to make a minor mistake in your code (a typo in a variable name, for example), then run the program and you will see Assistant in action.

B.3 Python

Both editors described above include Python in their installation packages. But for some reason you may want to install Python separately.

If you are on macOS or Linux, your operating system already have Python installed. Most likely it is not the latest version of Python, but it's not a problem at all: all programs we develop in this book will work with Python versions starting from 3.5. No need to install anything on these operating systems—at least not for this book.

If you are on Windows you will have to go to the official Python site: <https://www.python.org/downloads/windows/> and download the installer from there.

Please read the notes carefully (see the screenshot below) and choose the right Python version for your Windows version.

Python Releases for Windows

- [Latest Python 3 Release - Python 3.11.0](#)

Stable Releases

- [Python 3.11.0 - Oct. 24, 2022](#)

Note that Python 3.11.0 cannot be used on Windows 7 or earlier.

- Download [Windows embeddable package \(32-bit\)](#)
- Download [Windows embeddable package \(64-bit\)](#)
- Download [Windows embeddable package \(ARM64\)](#)
- Download [Windows installer \(32-bit\)](#)
- Download [Windows installer \(64-bit\)](#)
- Download [Windows installer \(ARM64\)](#)

- [Python 3.9.15 - Oct. 11, 2022](#)

Note that Python 3.9.15 cannot be used on Windows 7 or earlier.

- No files for this release.

- [Python 3.8.15 - Oct. 11, 2022](#)

Note that Python 3.8.15 cannot be used on Windows XP or earlier.

- No files for this release.

- [Python 3.10.8 - Oct. 11, 2022](#)

Note that Python 3.10.8 cannot be used on Windows 7 or earlier.

Another way to install Python on Windows is to open a PowerShell window and type `python`. Windows will suggest to install the right version of Python. You just have to accept it.