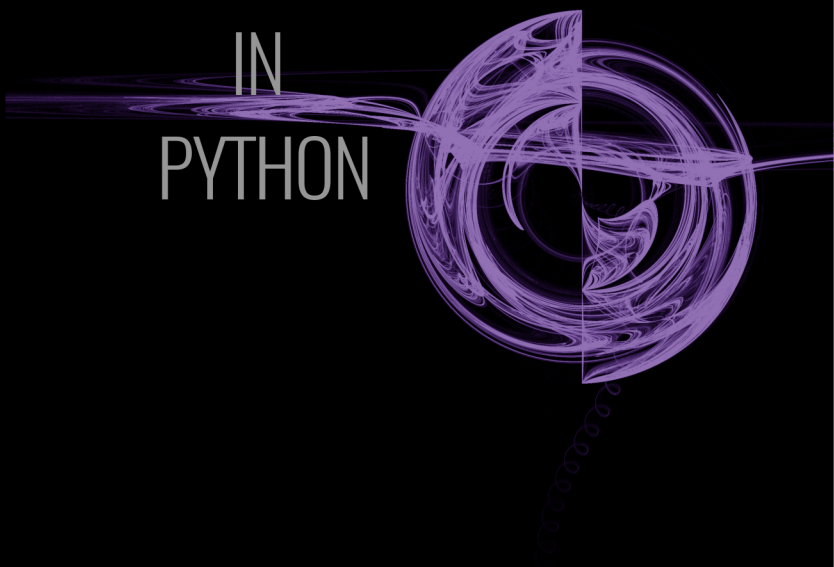
100 DAYS OF CODING

IN PYTHON



GIULIANA CARULLO

100 Days of Coding

In Python

Giuliana Carullo

This book is for sale at http://leanpub.com/100daysofcoding

This version was published on 2020-03-22

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Giuliana Carullo

Code Reviews 101

Technical Leadership

Identity Management 101

Stop Procrastinating

pip install python

Contents

Preface

Welcome to an intense session during which you will learn Python. This book will be your friend for the next 100 days to help you along the journey of learning the fundamentals of Computer Science and how to program them in Python.

This book will provide you step by step focused daily theory and practice to help you master:

1. programming foundation in Python;
2. main algorithms concepts;
3. popular data structures that you will encounter often in your life as a programmer;
4. design patterns that will help you in writing marvelous code.

Why this Book?

Sure enough plenty of books have been written on Python and the official documentation [PSF] is - and always will - remain the source of knowledge. A go to for doubts.

However, this book has been specifically designed to walk you through - step by step - not only Python syntax but also fundamentals of computer science that will make you a well-rounded programmer.

Who This Book is Meant For

If you are new to programming - in general - this book is right for you. Indeed it walks through the core concept of programming as well as detailing relative Python implementations.

Part 1 - The Basics

Introduction

This book is broken down into 5 Parts:

1. **Introduction**: covering basic concepts of Object Oriented Programming and their Python implementation.
2. **Algorithms**: including greedy approaches, dynamic programming and an introduction to np-hard problems.
3. **Data Structures**: explaining the main data structures that you will encounter in your journey as a programmer.
4. **Design Pattern**: providing the foundation of software architectures.
5. **Solutions**: for all the coding exercises presented.

The Study Plan

Where it All Begins

.

Fundamentals

Let's start with a general introduction to the Python language.

The goal of this Chapter is to give you the essential elements of the language and start getting used to Python code. In the first Chapters, indeed, what I am going to is to give you enough knowledge and hands-on practice - by means of excercises - to start writing basic code that you'll find in real life.

This is in line with the general idea of this book: it is not meant to deal with all the details and shades of the language, yet gently walk you through - in the shortest amount of time possible - your first programs in Python.

The remaining of the Chapter assumes you have your environment set up (i.e. python installed) and you know basic terminal interaction.

If you are not completely new to either Python or programming in general, you might want to quickly glance at this Chapter or completely skip it altogether depending on your knowledge.

The beloved hello world

Yes, it is kind-of a tradition to start every programming language with an hello world. Yet, from a simple excercise you can start getting confortable with the syntax. Before learning to drive a car, you need to sit into one. And this is what we'll be doing in this Section. We'll - indeed - walk through different ways to run Python code.

In Python, the code to print 'Hello, World' is:

```
1 print('Hello, World')
```

The command line

To execute our code let's open our terminal and run first the command

```
1 >>> python
```

This command allows you to run - line by line - your code.

Type in the above hello-world code and look the magic happen!

Just kidding, let's warm up!

.py code

Command line works for quick few lines testing. In any real scenarios you'd want to write code that you can run, re-run and reuse. So let's save our code in a file.

1. Create a file with .py extension (hello.py)
2. Open the file with the preferred editor (or IDE - e.g., Pycharm)
3. Type in the example and save
4. Open the terminal, navigate - if needed - to the folder that contains your hello.py
5. Run the code by

```
>>> python hello.py
```

Different process, same output.

The main

main() is a function provided by most programming languages with the purpose of defining where the program starts. Generally, this special function has a specific signature that defines it's inputs (arguments) and output (return).

Python automatically runs programs line-by-line scripts from the top to the bottom. However, the language still provides a way to alter the flow of execution. One the most basic is, as you figured, main.

Back to our example.

Create another file named hello_main.py and type in the following code.

```
if __name__ == "__main__":
    print("Hello, World")
```

If no typos or syntax issues exist you will get the same result as from the previous examples.

Coding Practice of the day

Warm up

Play around with the code provided in this Section. Change indentation, remove parts of the code and get familiar with the error messages you get out of the changes.

Medium

Try to fix the following code:

```
1  if __name__ == "__main__":
2      print("Hello, "World"")
```

The expected result is the print of the line: Hello, “World”.

See you tomorrow!

Basic Input/Output

We started, with a simple excercise, to learn how to print a simple sentence (string) with the print command. However, in most common situations you might want to concatenate several sentences, maybe numbers, etc. Hence in the following we'll learn some basic manipulation and how to implement it.

Output manipulation

String concatenation

Consider the example where you want to print two separate words: "Hello, " and "World".

The easiest way to achieve it by using the "+" operator as follows:

```
1 print('Hello, ' + 'World')
```

String literals

Sequence escaping

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Give it to me!

We learned how to print, let's get our feet wet with getting some custom-input as well. The most basic tool that Python gives us is the input() function.

Coding Practice of the day

Warm up
TBD

Medium
TBD

Power-up
TBD

See you tomorrow!

Basic Types

Numeric Types

Basic numeric types in Python are:

- integers
- float
- complex

Integers are finite numbers, either positive or negative, without decimals.

```
1 num_int_pos = 42
2 num_int_neg = -42
```

Float numbers are numbers, either positive or negative, with decimals. Python also supports the scientific notation, as show in the following snippet (line 3).

```
num_float_pos = 42.0
num_float_neg = -42.0
num_float_scientific = 42e2
```

Just as a reminder, the scientific notation - which is indicated with an *e* or an *E* - defines the power of 10 of the given number. In the example above, we declared a scientific number which is the power 10 of 42.

Putting all together, run the following code to match numbers to their relative *type* counterpart.

```
numbers = (42, -42, 42.0, -42.0, 42e2)

for n in numbers:
    print(type(n))
```

The output will look like:

```
>>> <class 'int'>
>>> <class 'int'>
>>> <class 'float'>
>>> <class 'float'>
>>> <class 'float'>
```

Even if you will encounter them way less often in your day to day programming, it is worth to mention that Python also supports **complex** numbers. In mathematical terms, a complex number is one that has a *real* part and an *imaginary* one. This type of number can be expressed as:

```
num_complex = 42+1j
```

In this example, 42 is the real portion, while the *j* indicates the imaginary counterpart.

Strings

A strings is a sequence of characters. In Python, strings are declared by wrapping characters either with a single or double quotation mark.

Hence:

```
var_string = 'hello Python!'
```

is the same as:

```
var_string = "hello Python!"
```

even if I recommend maintaining consistency of the notation you want to use in your code for the sake of uniformity and readability.

Other types

Coding Practice of the day

Warm up

Write the Python code that counts the number of character in a string. Print the result.

Medium

TBD

Power up

TBD

See you tomorrow!

Structure of a Python Program

I know, I hear you. We learned already about Python, but we want to get our teeth in our first program. Let's do it. Just few more concepts to go.

Statements and Expressions

A **statement** is an instruction that the interpreter (in case of Python) can execute.

Assignment, as we have seen so far, is a specific type of statement called: **expression**. Generally speaking, other example of expressions are:

- any arithmetic operation (e.g., sum, subtraction, division)
- operating with strings (e.g., concatenation)
- calling a function

For example:

```
1  # summing first four numbers is an expression
2  sum = 1 + 2 + 3 + 4
3
4  # string concatenation is an expression
5  signature = "name" + "surname"
```

A statement could be mistakenly defined as: a single line of code. But it not 100% true. I can be a single line of code as in the case of *assigment* (and often it is) but it can also be a more complex/long instructions. Statements can, indeed, be **multi-line**, even the simple ones. For example:

```
1  # single line statement
2  signature = "name" + "surname"
```

has the same outcome of:

```
# multi-line statement
signature = "name" + \
"surname"
```

As well as:

```
# yet another multi-line statement
signature = "name" +
/ "surname"
```

The istruction is the same, but it is scattered on several sequential lines. The first multi-line variations is reffered to as **explicit line continuation**, while the second one is the **implicit** variation.

Functions

A set of statements that are functional to executing a logically coherent piece of code can be grouped together to form a **function**. The general shape of a *function* in Python (pseudo-code) is:

```
# general function
def foo_bar(params):
    statement1
    statement2
    ...
    return "done"
```

Functions are composed of:

- the **signature**:

  - the **def** keywork
  - the name of function (e.g., foo_bar)
  - a variable number of parameters (in our example *params*). Parameters are used to provide **input** to the function.
- the **body** of the function, i.e., the set of statements that all together express the logic
- **return** statement:
  - in the form *return value*
  - used to define an output to the function
  - can be optional

Modules

Modules are the basic way of organizing code in Python. They are an arbitrary composition of:

- variables
- functions
- classes

The following snippet showcases the simplies module on earth: *hello.py*.

```
name = 'Giuliana'

def hello():
    print('Hi {}, my dear! '.format(name))

if __name__ == "__main__":
    hello()
```

One of the key benefits of using modules - other than simply structuring our code - is to write them once and reuse them every time it is needed to.

Consider another module, *multiple.py* that wants to greet an arbitrary number of times instead of just one. This cheerful module can resuse *hello.py* as follows.

```
import basics.modules.hello as greet

def multiple_hello(number:int):
    i = 0
    while i < number:
        greet.hello()
        i+=1

if __name__ == "__main__":
    number = input("How many times? ")

    #converting number from string to int
    number = int(number)
    multiple_hello(number)
```

input() function

This function allows to read from keyboard. Specifically, it displays the message and then waits for user's input. *input()* returns the *string* representation of anything typed before hitting *enter* when the message is promped.

The *import* function might still be fairly obscure to you at this point in time but…no worries! We will go deeper on it in this Chapter.

Packages

Simply speaking, a package is a way to group several modules together. Why do they exist?! Well imagine having all your files in your lapton in a single massive folder… confusing isn't it?

Packages are a way for structuring your code (modules) in a logically cohesive way and, hence, creating a hierarchical organization of your code.

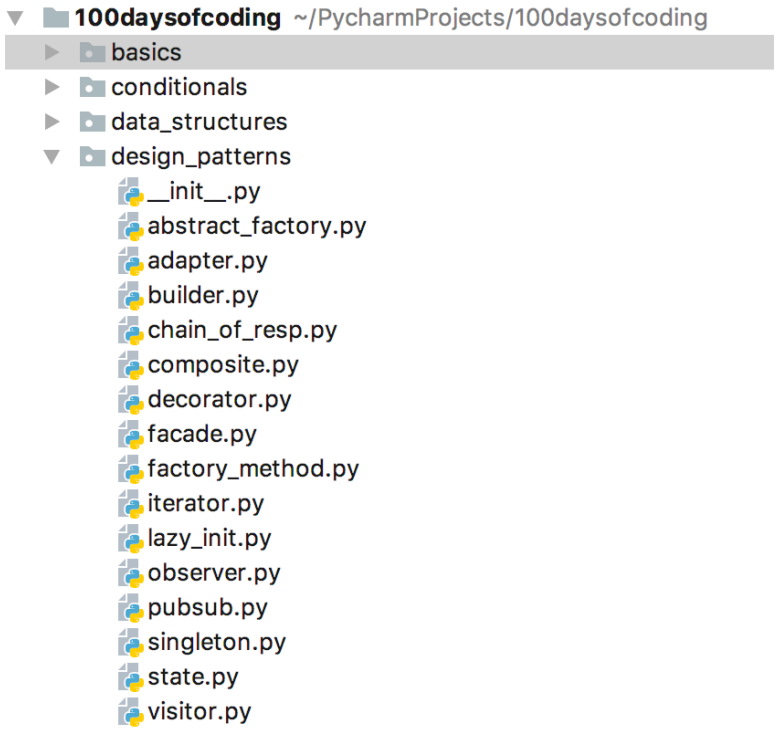


Fig x.x - Package Structure

Pretty simple, right?

Modular Programming

Other than simplifying our lifes from cahotic code, functions, modules and packages allow us to implement **modular programming**. There are plenty of benefits from using this approach:

- **Scoping**, which is where variables and function live (known as **namespace**). By breaking down the *visibility*

of our variables, functions and modules allows for better collision avoidance between identifiers.

- **Reduced Complexity**, analogously to the folder example. By breaking down the code into small pieces, it is easy to focus on a single aspect of the problem at hand.
- **Reusability**, which is the capability to write the code once and reuse it twice. Without modular programming, you would have a gigantic single block of code. What if you want to reuse few lines of code? Copy-paste. Duplicating code is not only painful but also a bad practice (we will get more into details on reusability later on in this book).
- **Mainteinability**: by logically splitting the code in cohesive smaller pieces and by sprinkling on top of it few more good practices (e.g., low cohesion, which means maintaining as minimum dependencies between different software components), the code will result less error-prone as well as easier to change, expand and reuse over time.

Import

We spoke about scoping and reusability. One limits visibility the other allows for reusing code. But how do we use code with limited visibility? Import!

Let's deal with the simplest import on earth, shown in the following Figure.

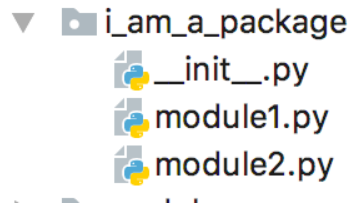


Fig x.x - Simple import

Where *module1.py* is

```
1  print('Imports are so cool!')
```

and *module2.py* is

```
1  import module1
```

The output from the above code is:

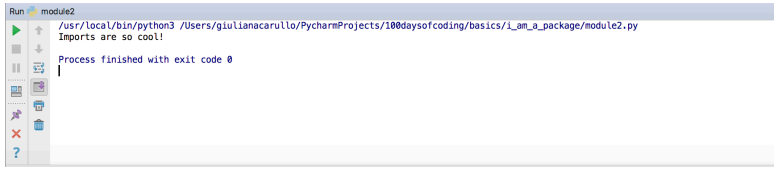


Fig x.x - Simple import: result

Mission accomplished!

Absolute vs relative

Importing a package

Putting it all together

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Variables

If you are completely new to variables, let's kick start this discussion with a definition:

> A **variable** is a mechanism used to store or reference data that can be:
>
> 1. Initialized
> 2. modified over time
> 3. reference memory locations
> 4. dismissed

In other words we can define a variable as:

> a simbolic (i.e., human readable) representation to a location in memory

Creating a variable in Python is super simple:

```
1  my_first_var = 42
```

If you already have experience with other programming languages, let's use your knowledge as a reference to better understand what Python is doing here. In Java, for example a variable can be declared as:

```
int my_first_var = 42
```

The first big difference between Python and other languages (Java, C, C++ included) is that there is no need for declaring the type of the variable. When creating a variable in Java, the following steps are executed:

1. hey! that is a int: let me allocate just enough memory to store an integer
2. let me reference that location
3. cool, now I have to save 42 at that localtion

The main consequence (and difference) is that types are way more strict in other programming languages. Hence a variable in Python can change over time:

- in terms of data (likewise all the other programming languages)
- in terms of type (peculiarity of Python)

Indeed in Python something like:

```
my_first_var = 42
my_first_var = "I am now a string!"
```

is totally fine.

Referencing

Let's go deeper into what happens with variables. A variable is a **reference** to an **object** in memory. The object might be a basic type - e.g., an *integer*, a data structure, e.g., a *list*, or a custom object, i.e. defined with the usage of *Class*.

```
my_first_var = 42
my_second_var = "my_first_var"
```

In the above example, my_second_var points at the same memory object as my_first_var.

For those of you with a background in *C*, the above code is similar with what happens with *pointers*.

```
    int my_var = 20;

    //declaring a pointer
    int *ptr;
    ptr = &var;
```

It is evident, however, that in *C* the pointer, i.e. *ptr* has to reference the same data type of the variable it is pointing to. Python is more flexible and a variable can reference any type at any point of its life. A blessing and a curse (for good coding).

Keywords

Keywords are words reserved to Python. Yes: Python can. And you can only used them as Python meant them to be. The following is a list of keywords.

and	as	assert	break
class	continue	def	del
elif	else	except	False
for	from	finally	global
if	import	is	in
lambda	None	nonlocal	not
or	pass	raise	return
True	try	while	with
yield			

For now, you just have to know that you cannot use them for: * identifiers, * function names, * class names, * you name it.

What they do will be covered in the various Sections of this book, so keep reading and learning. ## Naming Conventions

Naming convention might deserve - as a topic - an entire chapter - and I did in *Code Reviews 101*. However, I'll try to keep your brain nice and quiet on the topic.

First and foremost, don't skip this section, a bit wordy for sure (we want code! I hear you!). But they do exist for a good reasong: readability.

What To Avoid

Rule number 1: Don't use keywords as a variable name. Rule number 2: don't use keywords as a variable name. You got it.

Convensions

Here are the basics convensions for Python:

- **Package and Module**: all lower case, crisp and short. If more than a single word, use an underscore (_) as a separato.

- **Classes**: CamelCase. If you hate Java you can refer to it as CapWords as well.
- **Variables**, **Functions and Methods**: as for package and modules. Although **mixedCase** can also be used. Which one to pick? Base the decision on: consistency.
- **Global Variables**: don't use them. If you really really (really) have to use them (minimal scope), the follow the same rule set as normal variables.
- **Constants**: easy to spot in the code: all caps. As an example:

```
1 CONSTANT= 'I am a constant'
```

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Python Objects

Class

Initialization

Attributes

Methods

Functions

@properties

Properties provide a *pythonic* way for accessing and setting variables (i.e., getters and setters).

Let's start with an example.

```
class Book():
    def __init__(self, title):
        self._title = title
```

Suppose that the code above needs to integrate with another application that gathers titles in. Let's also suppose that at a certain point this external applications puts a limit on the length of the title: cannot be more that 20 characters (a bit of a weird constraints but... hey! What can we do?!)

So we change our code such that we can add constraints when setting our title:

```
class BookConstraints():
    def __init__(self, title):
        self._title = title

    def get_title(self):
        return self._title

    def set_title(self, new_title):
        if len(new_title) > 20:
            raise ValueError("Title '{}' is too lon\
g".format(new_title))
        self._title = new_title
```

Looking at the execution:

```
if __name__ == "__main__":

    book_with_constraints = BookConstraints ("pip i\
nstall python")
    book_with_constraints.set_title("pip install --\
target=your:brain python")
```

We get the following output:

```
Traceback (most recent call last):
...
ValueError: Title 'pip install --target=your:brain \
python' is too long
```

It worked! But we have the pythonic way for achieving this result.

```
class PythonicBook():
    def __init__(self, title):
        self._title = title

    def get_title(self):
        return self._title

    def set_title(self, new_title):
        if len(new_title) > 20:
            raise ValueError("Title '{}' is too lon\
g".format(new_title))
        self._title = new_title
        print("This is magic!")

    title = property(get_title, set_title)
```

Observe that, the last line makes a property object *title*. The actual operation happens on the private *title, however _title* (public) constitutes an interface to it.

If we run the code:

```
if __name__ == "__main__":

    pybook = PythonicBook("pip install python")
    pybook.title = "pip install python"
    pybook.title = "pip install --target=your:brain\
 python"
```

we get the following output:

```
This is magic!

Traceback (most recent call last):
  ...
ValueError: Title 'pip install --target=your:brain \
python' is too long
```

where the first set passes the check hence printing *"This is Magic"* and the second one triggering our error.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Polymorphism

Polymorphism is a foundational concept in Object Oriented programming. It refers to the ability to access the same method that - depending on the object is called upon - presents a different behaviour/output.

A simple example of application of polymorphism is given by the *built-in* function *type()*.

```
print(type(1))
print(type('hello'))


>>> <class 'int'>
>>> <class 'str'>
```

Inheritance

Inheritance is a way to implement polymorphism. Specifically, it enables an object to embed and potentially extend or redefine properties of another object.

The inherited class is called:

- Super Class,
- Base Class or
- Parent Class.

On the other side, the object inheriting from the super class is called:

- Sub Class,
- Children or
- Derived Class

Inheritance can also be expressed as:

> a Sub class *implements* a Super Class.

A simple example of inheritance in the following.

```
class Car():
    def move_front(self):
        return 'move front'

    def move_back(self):
        return 'Move back'

    def brake(self):
        return 'Break'

class Berlina(Car):

    def shape(self):
        return 'I can do everything a car can, but \
with a different shape!'

if __name__ == "__main__":
    car_generic = Car()
    print ('I can: {}, {}, {}'.format(car_generic.m\
ove_front(),
                                      car_generic.m\
ove_back(), car_generic.brake()))
    car_berlina = Berlina()
    print('I can: {}, {}, {}. \n{}'.format(car_gene\
ric.move_front(),
                                           car_berl\
ina.move_back(),
                                           car_berl\
ina.brake(),
                                           car_berl\
ina.shape()))
```

Logically speaking it can be applied when an *'is'* relationship

can be applied between *subclass* and *superclass*.

This is...Super!

We saw so far how to extend a class. However, we implicitely used another method: *super()* (lines 21/22 of the previous snippet).

As you might have noticed, we did not defined any *brake()* method for our Berlina class, whilst the *Car* still brakes.

This situation can be made explicit by the following Python code (not needed, just for the purpose of explaining what is happening).

```
1  class Berlina(Car):
2
3      def shape(self):
4          return 'I can do everything a car can, but \
5  with a different shape!'
6
7      def brake(self):
8          return super().brake()
```

Its general form (Python 3) is:

```
1  super().methodName(args)
```

Overload

Super() also becomes very handy when we want to provide an enhanced version in a subclass for a specific function.

For example:

```
class CustomBerlina(Berlina):

    def __init__(self):
        self._color = None

    # an example of overloading
    def shape(self, color=None):
        super().shape()
        self._color = color
        print('Now I am {}'.format(color))
        return 'Always a Berlina, but I am {}'.form\
at(self._color)

if __name__ == "__main__":
    car_berlina = CustomBerlina()
    print('I can: {}, {}, {}. \n{}'.format(car_berl\
ina.move_front(),
                                           car_berl\
ina.move_back(),
                                           car_berl\
ina.brake(),
                                           car_berl\
ina.shape('red')))
```

What we did is **overloading** a method of the superclass by adding *optional* arguments to the relative signature.

Override

For the sake of going through another concept (i.e., **override**), let's create a Limousine out of a Berlina.

```
class Limo(Berlina):

    # an example of override
    def shape(self):
        print('Just a little of a stretch')


if __name__ == "__main__":
    car_berlina = Limo()
    car_berlina.shape()
```

By overriding a method, we substitute the code in the parent class with what is specified in the subclass.

As shown in the example, the signature remains the same (and needs to be as such), whilst the method's code changed.

Congratulations! With a little of a stretch we transformed a Berlina into a Limo!

Types of Inheritance

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

Composition

Composition, differently from inheritance, is applied for *'have'* relationships between objects. A simple example of composition is shown in the following snippet.

```
class Windscreen():
    def __str__(self):
        return 'windscreen'

class Gear():
    def __str__(self):
        return 'gear'

class Car():
    def __init__(self):
        self._windscreen = Windscreen()
        self._gear = Gear()

    def __str__(self):
        return 'I have : {} and {}.'.format(self._w\
indscreen, self._gear)

if __name__ == "__main__":
    car = Car()
    print(car)
```

Rule of Thumb

A commonly applied principle when finding the right trade-off between inheritance and composition states that:

> to achieve polymorphism, composition should be preferred instead of inheritance.

Composition usually present a more flexible design, thus resulting maintainable in the long run. Not only design based on composition might be easier to write in first place. It will also accommodate future requirements and changes without requiring a complete restructuring of the hierarchy

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Conditional Flows

Conditionals are used to logically alter the *"normal"* path of the code.

The most commonly used conditionals are:

- *if statement*
- *loop* (that we will consider in next Chapter)

Conditionals use mathematical operations like:

- Equality: *a == b*
- Not Equals: *a != b*
- Less than: *a < b*
- Less than or equal to: *a <= b*
- Greater than: *a > b*
- Greater than or equal to: *a >= b*

to enforce the flow to change if a certain runtime condition happens.

As a simple example:

```
a = 10
b = 1
print(a==b)

b =10
print(a==b)


>>> False
>>> True
```

The above code acts exactly as its explicit version:

```
if a==b:
    print(True)
else:
    print(False)
```

If statement

This is also our very first example of *if statement*. Its general form is:

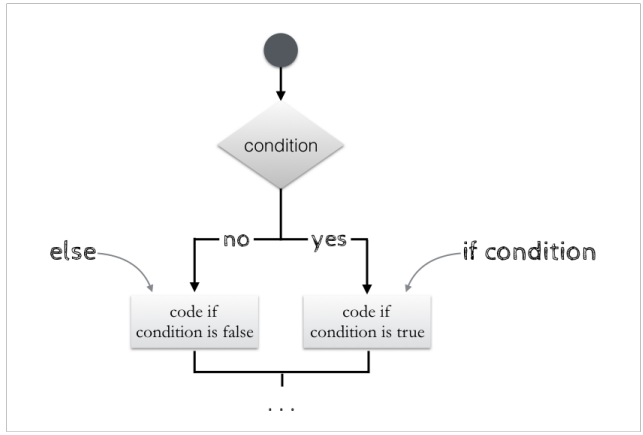


Fig n.n - Basic if flow

Relative pseudo-code:

```
if condition_is_true:
    print(True)
    # flow of the code if condition holds true
else:
    print(False)
    # flow of the code otherwise
```

Multiple Checks

The flow can also be further altered by using *elif*:

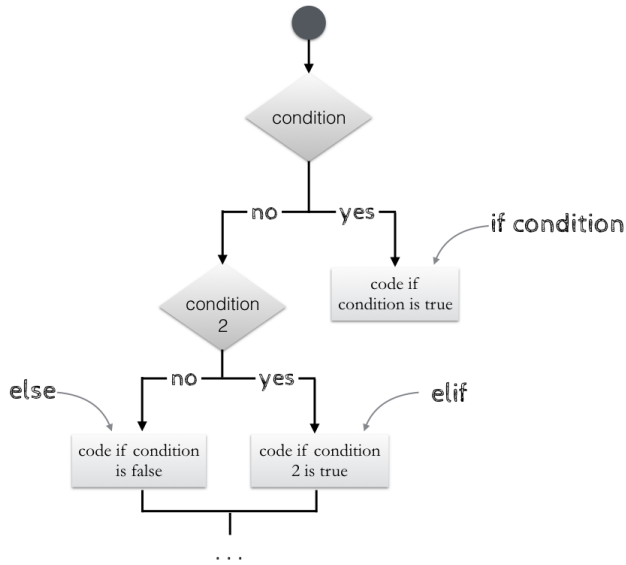


Fig n.n - Complete if flow

Relative pseudo-code:

```
if condition_is_true:
    print(True)
    # flow of the code if condition holds true
elif condition_2_is_true:
    print(True)
    # flow if condition_2 holds true
else:
    print(False)
    # flow of the code if both condition and condit\
ion_2 are not true
```

In summary, *if/elif* structure allows for a cascade of checks.

Python also supports *AND* and *OR* operators to perform multiple checks at the same time. This is very handy when,

for example, the same behaviour needs to be applied when - from the example above - either *condition* or *condition2* hold true.

In this case the previous snippet can be rewritten as:

```
if condition or condition_2:
    print(True)
    # flow of the code if condition holds true
else:
    print(False)
    # flow of the code if both condition and condit\
ion_2 are not true
```

Membership

Python also supports more complex checks for structured data, including:

- Membership: *in*
- Non Membership: *not in*

The form of usage is as follows:

```
if condition in data:
    # do something
else:
    # do something else
```

A simple example of using membership is:

```
i = 3
if i in range(5):
    print('I am included in the range!')
```

Coding Practice of the day

Warm up

Implement a Python program that, given in input a letter *l*, prints if *l* is a vowel or a consonant.

Medium

Implement a Python program that, given in input an object *o*, checks its type (string, integer or none of these).

Hint: study how *isistance()* work.

Power-up

TBD

Â {pagebreak}

See you tomorrow!

Loops

Loops are a way to apply common logic (blocks of code) multiple times until a condition is met.

The flow diagram for loops is shown in the following:

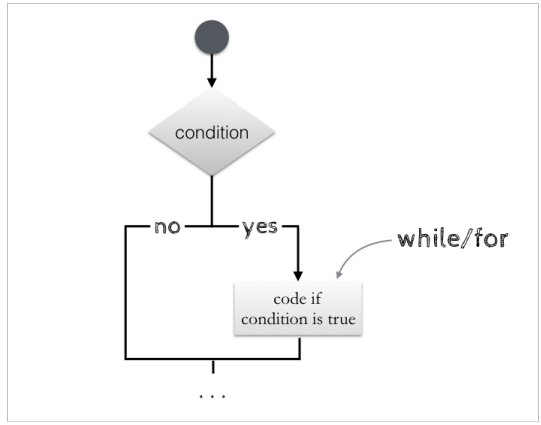


Fig n.n - Basic if flow

Python supports two types of loops:

- while
- for

While

The syntax for the *while* construct is as follows:

```
1  while condition:
2      # block of repetitive code
3      ...
```

For

The syntax for the *for* construct is as follows:

```
for condition:
    # block of repetitive code
    ...
```

Further Modifying the Flow

Break

Break and **continue** are two other statements to alterate the flow of a loop.

```
for i in range(10):
    if i == 4:
        print('Got bored: interrupting the loop')
        break
    print('Iter {} - Looping again!'.format(i))
```

The above code has as output:

```
>>> Iter 0 - Looping again!
>>> Iter 1 - Looping again!
>>> Iter 2 - Looping again!
>>> Iter 3 - Looping again!
>>> Got bored: interrupting the loop
```

Break interrupts the loop even if the condition, in our example *i in range(10)*, still holds true.

Continue

Continue allows to skip the current iteration if a certain condition holds true. Back to our previous example, using continue instead of break.

```
for i in range(10):
    if i == 4:
        print('Got bored: interrupting this iterati\
on')
        continue
    print('Iter {} - Looping again!'.format(i))
```

The above code has as output:

```
>>> Iter 0 - Looping again!
>>> Iter 1 - Looping again!
>>> Iter 2 - Looping again!
>>> Iter 3 - Looping again!
>>> Got bored: interrupting this iteration
>>> Iter 5 - Looping again!
>>> Iter 6 - Looping again!
>>> Iter 7 - Looping again!
>>> Iter 8 - Looping again!
>>> Iter 9 - Looping again!
```

Hence it only skips the fifth iteration.

pass

The *_pass* statement allows the code to get lazy and do absolutely nothing. Nothing! For example:

```
import time

start = time.time()
for i in range(5):
    # busy wait
    time.sleep(1)
    pass

print('I have been doing nothing for {} seconds.'.f\
ormat(time.time()-start))
```

The output would look like:

```
I have been doing nothing for 5.01263427734375 seco\
nds.
```

Differently from break and continue, pass can be used also within a method or function to enable it to do (again) nothing.

```
def feeling_lazy():
    pass
```

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Error Handling

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Documentation

Spoiler alert: documentation and comments - done for good - matter. Mumble, mumble.

This Chapter will go streight to the point: how to write good comments to support your colleagues and… the future version of you! We will walk through some of the bad practices and provide guidance on how to write documentation…for good.

The Syntax

Python, analogously to other programming languages, supports two types of comments: * **in-line**, achieved by placing a # character before the actual comment (one liner) * **extended documentation**, achieved by wrapping multiple lines of text within """.

We already saw *in-line* comments so far, but for the sake of completeness, here it is an example of this style of comment.

```
1 while condition:
2     # I am an in-line comment
3     ...
```

An example of extended documentation is in the following snippet.

```
1 """
2 I am an extended documentation.
3 In this case I document the following function.
4 Mumble mumble.
5 """
6
7 def mumble_mumble():
8     pass
```

Extended documentation is also refferred to as *docstrings*.

If the Code is Good I don't need Comments Philosophy

Someone approaches the philosophy of *'if the code is good, I don't need comments'*. Which is partially true, having a good piece of code surely is the way to go (at the end of the day, well-written code is all this book is about). However, this philosophy might be applicable for a very small codebase.

As soon as the code grows in size, you might end up in reading the code line by line for each and every piece you need to build upon. What if the code is well written but still implements complex algorithms? And if also naming is not in track, you are setting your-future-self and your colleagues for failure.

Far away from recommending pages-long comments, but they should be in place and well written (as described in the following Sections), even for good code. As a plus: people will complain about not well documented code, they will not do it for more verbose descriptions.

Conditions and Flows

Comments serve the purpose of specifying how the method-/procedure performs its task. They are not meant to punish so-so code writers. Comments are a way to succinctly abstract the inner work of your code in common language. And supplementing comments with pre- and post- conditions is beneficial.

A **pre-condition** is defined as a set of conditions that need to hold true when the method or function is called.

A **post-condition** is a statement describing the condition after the method or function has performed its duties.

Embedding them into your comments is not only beneficial from a readability perspective, but it will also help during test design. Consider, for example, the scenario in which you have a subset of the team strictly focusing on designing and developing tests. In this case, it will be surely cost saving for them to have specified exactly the conditions that need to hold true before and after a call. It will be definitely faster and easier for them to check for edge cases, possible errors and design overall better and more complete test cases.

Clear pre- and post-conditions also helps in defining the **flow** (i.e., the logic) of the block of code. What to expect? Documenting the flow should also include any raised Exception. Yes, please, I definitely want to know if the code I am using might go 'boom' at a certain point.

IO Definition

Comments should clearly document the type of input(s) and output(s). This is especially true for dynamically typed languages like Python. Let's look at the following code snippet. It is an example of both so-so naming and confusion created by missing defined types into the documentation.

```
1  def add(first, second):
2      #method body
3      ...
```

It is pretty clear that the method is performing an add operation. However, it is not clear the type of the inputs. Are

these operands? Are they integers? Float numbers? Strings? Is the method performing more complex operations on data structures? The parameter first might be a dictionary, a list, or anything else, and it might be performing more complex operations to add the parameter second to it. Or they might be both data structures and the method is adding the elements from the second data structures to the first one.

Yes, you might read the code to understand the inputs' types. Is it the most effective solutions? In this case, the snip is simple, so yes it takes short time to check it out. Otherwise, we are back to the consequences of the 'If the code is good, I don't need comments' philosophy.

You might also think: 'well… if I improve the names of the parameters (first and second), I can avoid writing comments?'. The answer is still: probably no. What happens to the abstraction power of the method? What about its reusability? It might be context-dependent, of course. But the rule of thumb is, clearly state the IO's types.

Inline Comments

Too many inline comments instead of - as an example - a clear comment on top of the method, drive me crazy. This does not mean to avoid inline comments at all costs. They should be present where actually needed. Anyway, when a method can be effectively described with docstrings, I would definitely go for this option.

From time to time, too many inline comments might also hint other issues within the code. Indeed, it might be the case of too complex, too long, more than a single feature implemented by the code. When it is not easy to shortly and

effectively describe the code you wrote, check again the code. Is it implementing just one feature? Is it too long? Might it be split? If so, you have more than a single defect to fix.

TODOs

Inline comments are also used to signal a block of code that needs to be debugged, tested, or generally improved. In other words, they are the handy TODO and FIX-ME comments we all introduce in our code to avoid breaking the process of writing our most creative, productive piece of coding art.

A nice piece of craft could be:

```
1  def my_magnificent_method():
2      # TODO: comment me, Giuliana said so
3      ...
```

Even if I would love to read such code, please don't wait a decade before fixing any TODO and FIX-ME. They are meant to be temporary, and they should remain so. Even more, no TODOs and FIX ME in the code you are somehow releasing.

That's Obvious

Writing comments is good. Writing good comments is tremendous! However, commenting also obvious things is not appropriate. If the code is really simple, there is not need for commenting each and every line of it. Consider the following example:

```
...
 # gets the glossary of the book
 glossary = book.get_glossary()
 # check the keywords we are looking for
 if keyword in glossary:
     # return the associated definition
     return glossary[keyword]
 else:
    # return None since the keyword is not in gloss\
ary
    return None
 ...
```

It is obvious what the code is doing, and good variable naming helps. A lot. A better commenting style would be:

```
...
 # Returns the definition for a keyword if it exist\
s.
 # None otherwise.
 glossary = book.get_glossary()
 if keyword in glossary:
     return glossary[keyword]
 else:
    return None
 ...
```

It is much more readable, isn't it?

Did you just lie to that programmer?

Comments are intrinsically virtuous and worthy of reward. They really are. Writing good comments such that it is easy for a programmer to understand the functionality and just glance over it, whilst lying is a big no no.

If you write top notch comments, but you lie in them I appreciate your sense of humor. I really do. But please, don't.

Comments Driven Development (CDD)

Comments Driven Development (CDD) is centered on writing plain text description of the code first, then the actual implementation. They help in brainstorming the solution, setting clear inputs and outputs and describing the kind of functionality is going to be implemented.

As an analogy, think about teaching something to someone. In order to properly express in natural language a certain concept, you have to have a good understanding of it. The same happens with CDD: you are able to write the comment only if you properly thought about what that piece of code is going to implement.

Furthermore, often times we are so inspired that we start compulsively typing lines and lines of code, to discover, only in the end, that we have to travel back in time to remember what they were supposed to do.

Even if I don't think about implementing CDD is an absolute must, it can be handy.

Coding Conventions

Some coding conventions like PEP8 [Python Software Foundation] for Python, impose the presence of docstrings only for public modules, functions, classes, and methods. They remain optional for private or protected, where usually a comment after the function signature still describe the method.

As soon as you become aware of the language you are going to use for your project, check coding conventions and stick with them.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Part 2 - Algorithms

Introduction

So far, we learned the main elements of Python's syntax and the logic behind them: what they do, how they are used.

However, syntax means nothing if not in perspective of solving problems.

An algorithm, even before writing a single line of code, can be defined as a mean of solving logically a problem.

More generally, an algorithm can be used to solve and be applied to a variety of applications.

In this Chapter, we'll learn:

- general knowledge behind algorithms, including how performances are defined;
- some classes of algorithms that you migh encounter in your life as a programmer.

You might be thinking: why do I even need to know algorithms at all?

Spoiler alert: well, most interviews ask about algorithms.

Jokes apart, some applications might not be of obvious application at first glance, whilst they can be used in many real world scenarios.

Recursion

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Iteration

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Greedy Agorithms

Thinking Greedy

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Dynamic Programming

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

NP-Hard problems

We wrote enormous quantity of code used, re-used and misused since the first programming language was born.

Yet, so many problems are NP-Hard. Isn't it fascinating?

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Part 3 - Data Structures

Introduction

Welcome to an introduction to one of the most beautiful pieces of art out there: data structures!

Intro to Data Structures

Data structures are used into any single piece of code out there, even the simpliest - yet useful - one. And they do play an important role into building up your coding skills. When picking a choosing them a lot of trade-offs and scenarios need to be considered. And when properly implemented into the context of your code they allow for the following benefits:

- Readability: using appropriate data structures can allow you to not overcomplicate code hence improving how easy to read and understand code is.
- Better overall design. Starting from project requirements thinking about data structures is fundamental and it allows for well organized data, improved performances both in terms of time and memory. Worth to stress further that, performances should not be an after though into the coding process. As a consequence, the more suitable the data structure is, the more optimized, effective and efficient your code will be as a result.

Main data structures are summarized in the following table, together with their relative average time complexity for basic operations, including: access, insert, delete, search).

Data Structure	Access	Insert	Delete	Search
Array	O(1)	O(n)	O(n)	O(n)
Linked-List	O(n)	O(1)	O(1)	O(n)
Doubly Linked-List	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(1)	O(1)	O(n)
Stack	O(n)	O(1)	O(1)	O(n)
Hash Map	Constant (avg)	O(1)	O(1)	O(1)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)

Table 3.1 - Time Complexity (average)

Array

Arrays are one of the easiest and most used data structures. It consists of a collection of elements that can be accessed by means of their position, namely *indexes*. Arrays can be *linear* (i.e. single dimension) or *multidimensional* (e.g., matrixes).

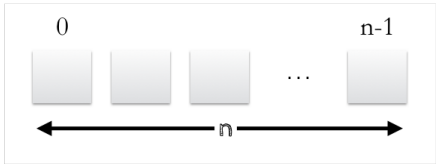


Fig 3.2 - Array

Arrays are generally fixed in size (i.e., how many elements they contain), which is defined during initialization. They are homogeneous data structures. In other words thet do support only a single type (e.g., all integers).

Some languages support dynamic sizin as well as different data types. In Python, these variation of the classic array is called a *list*. The underlying implementation of a list differs from arrays and in the remaining of the section we will use the more flexible list for our examples. In Python, lists are defined as:

```
1  array = [1, "string", 0.2]
```

and all the types are properly managed

```
type(array[0])
>>> <type 'int'>

type(array[1])
>>> <type 'str'>

type(array[2])
>>> <type 'float'>
```

They come with several pros and cons.

Th benefits of using this data structure comes from their easiness and the linear access to the elements as shown in *Table 4.1* – where *n* is the number of elements into the list.

Main cons is that inserting and deleting items from the list is expensive due to shifting operation required. The worst-case scenario, indeed is the following: consider deleting the first element of the list which is at index = 0. This operation creates an empty spot. To maintain index and ordering, all the items from index = 1 and following needs to be shifted one position backward. This gives us a complexity of *O(n)*. Analogous considerations are for insertion.

Internals

Coding Practice of the day

Warm up

Given an array *A* of integers in input, print it in reverse.

Example

A = [1,2,3,4,5]

Printed Result:

5 4 3 2 1

Medium

Given an array *A* of integers in input, reverse and print it without using builtin functions.

Example

A = [1,2,3,4,5]

Reverse:

A = [5,4,3,2,1]

Power-up

Given two arrays *A* and *B*, count the occurrences of *B* being a subset of the elements of *A*.

Example 1

A = [1,1,1,1,1]

B = [1,1]

occurrences = 4

Example 2

A = [0,1,1,0,1,0]

B = [0,1]

occurrences = 2

See you tomorrow!

Linked-List

A Linked-List is a collection of nodes, where each node is composed by a *value* and a *pointer* to the next node into the list.

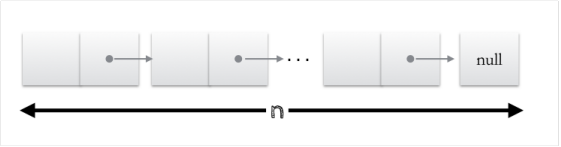


Fig 3.3 - Linked-List

Compared to arrays, *add* and *remove* operation implement a slightly more complex logic - yet since no shifting is required.

Specifically, removal an item from a list can be performed by changing the pointer of the element prior to the one that needs to be removed. The prior element will need to point to the element following the item that needs to be removed from the list Time complexity remains *O(n)* in the worst case.

Indeed, since we lost direct access by means of indexes provided by arrays, in order to find the item that needs to be removed, we need to navigate the entire list until we reach its predecessor. Removing the last element of the list represents the worst case.

Internals

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Doubly Linked-List

A Doubly Linked-List is similar to the previous one. However, each element (also called node or item) stores also a pointer to the previous element in the list.

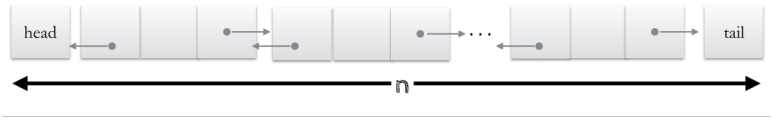


Fig 3.4 - Doubly Linked-List

As you can already imagine, adding and removal operation add another bit of logic complexity since two pointers (predecessor and successor) need to be updated when these operations are performed. As a consequence *add* and *remove* operations still take *O(n)* due to the sequential access needed to locate the right node.

Internals

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Stack

A Stack is a collection of elements that is managed with *Last In First Out* (LIFO) policy. Internal implementation varies and can be achieved using either Arrays or Linked-Lists.

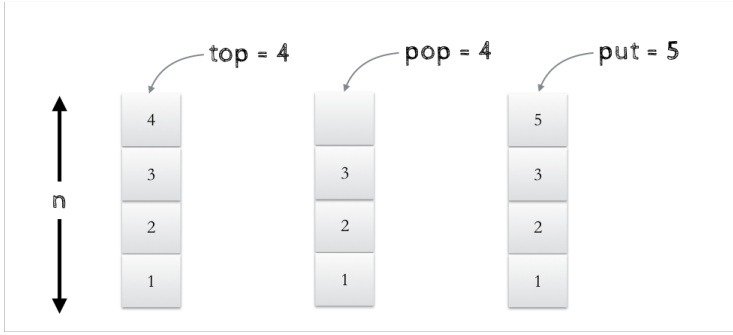


Fig 3.5 - Stack

The Stack data structures exposed three main methods:

1. top(), that returns the element on top of the stack,
2. pop(), that removes and returns the element on the top of the stack,
3. put(), that adds a new element at the top of the stack.

Some of the problems that mostly benefits from the utilization of this data structure includes pattern validation (e.g., well-parenthesized expressions) as well as general parsing problems due to the LIFO policy.

Internals

Coding Practice of the day

Warm up

Given an array *A* of integers in input, print its elements in revers by using a stack.

Example

A = [1,2,3,4,5]

Printed Result

5 4 3 2 1

Medium

TBD

Power-up

Implement a Stack class and provide the logic of main methods: top(), pop() and put().

See you tomorrow!

Queue

A Queue is defined as a collection of elements that is managed with *First In First Out* (FIFO) policy. Similarly to Stacks, it can be implemented either with Arrays or Linked-Lists.

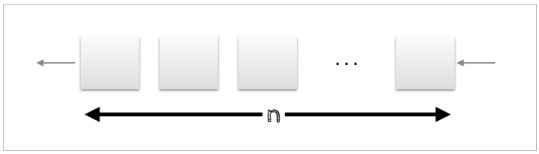


Fig 3.6 - Queue

Queues provide three main methods, in a similar fashion to what is performed by Stacks:

1. peak(), that returns the first (figurative left to right) element in the queue,
2. remove(), that removes the first element and returns it,
3. add(), that adds a new element at the end of the queue.

Queues are often used in concurrent programming, where several tasks need to be executed as well as patterns for messages exchange. In broader terms, queues suit well cases where order preservation is necessary, hence benefiting from a FIFO approach.

Internals

Coding Practice of the day

Warm up

TBD

Medium

Implement main Queue methods using a Stack.

Power-up

TBD

See you tomorrow!

Hash Map

Hash Maps have a very different storage strategy for the data: each element is broken down into a (key, values) pair.

In spite of a more complex internal logic, using Hash Maps has the main benefit of providing insert, delete and lookup operation in *O(1)* time in the average case.

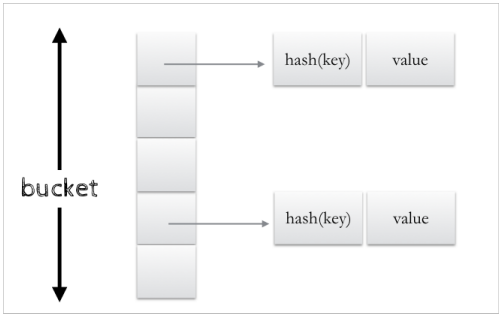


Fig 3.7 - Hash Map

As the name of this data structures expresses, keys are stored in locations (also referred to as *slots*) based on their **hash code**. Upon inserting a new element, a hash function is applied to the key, resulting into an *hash code* which is used then to place and associate the relative inputted value.

Different implementation can use different Hash functions. Yet they have a common characteristic: the same key must corrispond always to the one and one only hash code. A good hash function also must minimize cases where two different keys correspond to the same hash code, in which case it would produce a **collision**.

Worth also to notice that the benefits in terms of run time, strictly depend on the strength of the hash function. Indeed, one other characteristic of a solid hash faction is its capability to uniformly spread the entire set of available slots, namely

the **bucket** (e.g., an array of keys).

Hash maps have good performances in many real-world scenarios.

Internals

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

Â {pagebreak}

See you tomorrow!

Binary Search Trees

Trees is another fairly used data structure when modeling real-world scenarios.

Formally, a tree is a collection of elements, where each of them has a *value* and a variable number of *pointers* to other nodes (namely, *childs*). Nodes in a tree are organized in a *top-down* fashion, meaning that pointers usually link nodes from the *top* to thr *bottom* and each node has a single pointer referencing to it.

The note at the top is called the *root* of the tree. Nodes at the very bottom do not have *childs* and are called the leaves of the tree.

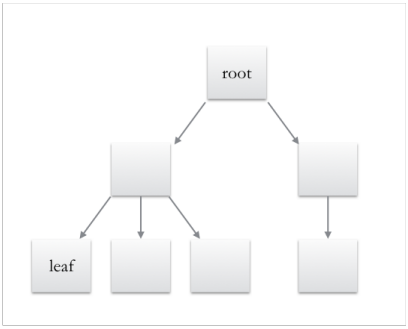


Fig 3.8 - Tree

A lot of problems require a particular type of tree: *binary tree*. What distinguish a tree from a binary one is that each node in the binary tree has at most two *child*, namely *left* and *right*.

When ordering is required, an extension of the Binary Tree is used: the *Binary Search Tree* is used. BS Trees are built by embedding ordering among its elements in such a way that, for every node:

- the left child's value is always less that or equal to the node's value;
- the right child's value is always greater than or equal to the node's value.

As also the name of this type of data structure suggests, the benefits occurring from its utilization emerge when a fast search of elements is needed.

This is enabled by the ordering enforced upon node creation, that allows for excluding half of the tree at every step of the search (in the *average* case).

In general terms, recursive problems are well modelled by trees.

Internals

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Takeaway

This Chapter on data structures is not meant to be a data structure book. It aims at providing you a quick yet useful introduction on using data structures in common scenarios.

I'd love to close this Chapter on data structures also providing you some indicators that can guide you when opting for one data structure in favore of another one:

- the size of data,
- how frequently data changes and its type,
- which operation are more frequent in your context.

Some rules of thumb:

1. do not force the API of a given data structure. If you are thinking about extending if with operations that are not natively supported, probably you'd be better off switching to a different data structure;

2. do not think only at the scenario you are facing at the moment. Think long term. This is way common when thinking about the size of the data and how it might evolve in the future.
3. use the 80/20 pareto principle to help you choose the best data structure. This is useful for optimization purposes. Which operation do you perform the most? Than generally needs to be optimized.
4. don't overcomplicate. More complex data structure still requires some more memory management and processing.

Further Reading

I love data structures. And amazing books are out there that dig deeper into various data structures and algorithms in general. Top of the list [Cormen et al.] is a must read if you haven't yet.

Part 4 - Design Patterns

Introduction

This Part of the book we will go through some general concepts around software architectures, specifically:

- common and widely used design patterns;
- does and don’ts;
- main issues you might encounter at design time.

Design Patterns

Let's shower a smelly architecture. One of the common ways to improve the design is by using appropriate design patterns. Patterns are simply a way to organize conceptual problems and relative code to solve common issues.

They are broken down into 4 main categories:

- **creational**, design patterns that aim at focusing on object creation;
- **structural**, that simplify and optimize the relations between different objects;
- **behavioural**, that establishes common pattern for communications between components;
- **concurrency**, that are specifically designed to address multi-threading scenarios.

In this Section, we will learn common patterns from the first three categories and shown in the following Table.

Concurrency patterns are a bit of a streatch and not part of the general goal for this book. However, foundations about concurrent programming will be discussed in the *Concurrency, Parallelism and Performances* Chapter.

Class	Patterns
Creational	Singleton
	Lazy Initialization
	Builder
	Abstract Factory
	Factory Method
Structural	Adapter
	Decorator
	Facade
	Composite
Behavioural	Observer
	Publisher-Subscriber
	Iterator
	Visitor
	State
	Chain of Responsibility

Creational

In this Section, we will explore some of the main *Creational* design patterns.

Class	Patterns
Creational	Singleton
	Lazy Initialization
	Builder
	Abstract Factory
	Factory Method

Singleton

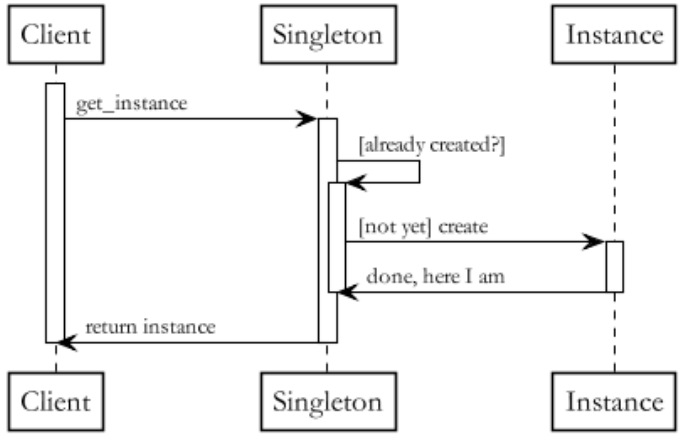


Fig 5.1 - Singleton

A **Singleton** is the easiest pattern you can learn. It goal it to restrict the number of instances of a class.

The objective is achieved by hiding the constructor of the sclass by means of declaring it *private*. A *get_instance()* method is used, instead, to create the singleton object. This method indeed will only instantiate the object upon the first very first call. Following calls will result in the method simply returning the instance previously created.

A Singleton might be useful when there is a need for shared yet controlled access to resources (e.g., concurrency). Generally speaking singletons are implemented when one - and only one - object needs to be created. A simple scenario includes logging.

The code

The following snippet of code provides a simple implementation of the Singleton pattern in Python.

```
class Singleton:

    def __init__(self):
        if Singleton.__instance:
            raise Exception("I am a singleton!")
        else:
            Singleton.__instance = self

    @staticmethod
    def get_instance():
        if not Singleton.__instance:
            Singleton()
        return Singleton.__instance
```

Don'ts

The Singleton has not the best branding and has been pointed out in few cases as a *'bad smell'*. One of the reason for it that if not properly implemented programmers can fall into the trap of the *'First lady'* smell. In other words, a Singleton - if misused - can result into a massive components that tries to do everything by itself. As almost everything on earth, I do not see a problem in the Singleton per se. Internet can be used for good or for bad. Singletons can be used good, or bad.

However, if you find yourself in growing the logic of your singleton more and more over time, you are probably using the wrong pattern for the problem that you are trying to solve.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Lazy Initialization

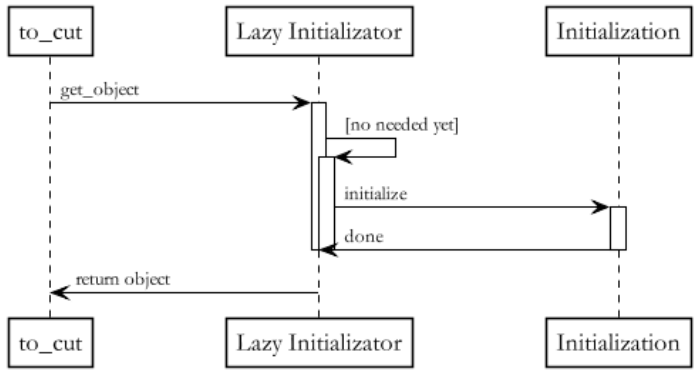


Fig 5.2 - Lazy Initialization

The goal of the Lazy Initialization pattern is to instantiate an object only when required - for real.

Instead of creating the object into the constructor, this pattern invokes the construction only when the object received the first call by means of one of its methods.

The main reasoning behind this pattern is that some logic that goes into logic creation can be computationally heavy. Hence, performances can be optimized by postponing it until the moment it is really needed to be performed.

Lazy Load is a common variation and implementation of this pattern applied to databases (DBs). Would you load the entire DB in advance and have it handy just in case it is needed? Nope, not really. Hence, the Lazy Load only retrieves the portion of data needed and stores it in memory when an actual computation is needed.

The Code

The following snippet of code provides an example of Lazy initialization in Python.

```
class MyHeavyObject:
    def __init__(self):
        self._heavy = []

    @property
    def heavy(self):
        if not self._heavy:
            print ('I am doing heavy computation.')
            # expensive computation
            ...
```

Don'ts

As for most of this patterns: do not overcomplicate your solution. Specifically, if you have a performance bottleneck (e.g., small and simple objects that are often accessed) do not use Lazy Initialization.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Builder

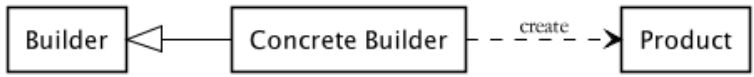


Fig 5.3 - Builder

The **Builder** breaks down the creation of a complex object into smaller separated creational tasks, alongside the *KISS* (Keep It Simple Stupid) principle. It

Specifically, the *Builder* pattern provides an expernal interface that allows for the creation of the complex object as a single task. However, it internally handles the breakdown on the big object into smaller objects using what are called *ConcreteBuilders*, each of which performing one step of the entire processing pipeline.

As a result, this pattern is helpful when the object to be built can be constructed via an arbitrary number of tasks.

The Code

The example code below shows the creation of a laptop. Virtually, a laptop can be broken down into several creational tasks that need to be performed in order to have the final product such as CPU, RAM, disk, etc.

```
# Abstract Builder
class Builder(object):
    def __init__(self):
        self.laptop = None

    def new_laptop(self):
        self.laptop = Laptop()


# Concrete Builder
class BuilderVirtualLaptop(Builder):
    def build_cpu(self):
        self.laptop.cpu ='whatever cpu'

    def build_ram(self):
        self.laptop.ram = 'whatever ram'

    ...


# Product
class Laptop(object):
    def __init__(self):
        self.cpu = None
        self.ram = None
        ...

    # print of laptop info
    def __repr__(self):
        return 'Laptop with cpu = {} and ram = {}'.\
format(self.cpu, self.ram)

# Director
```

```
34 class Director(object):
35     def __init__(self):
36         self.builder = BuilderVirtualLaptop()
37 
38     def construct_laptop(self):
39         self.builder.new_laptop()
40         self.builder.build_cpu()
41         self.builder.build_ram()
42         ...
43 
44     def get_building(self):
45         return self.builder.laptop
46 
47 #Simple Client
48 if __name__=="__main__":
49     director = Director()
50     director.construct_laptop()
51     building = director.get_building()
52     print building
```

Don'ts

The definition is fairly simple, thus stick with it, taking into account that embracing this design pattern has minor disadvantages including writing more Lines of Code (LOCs). A signal that a builder is not appropriate is when the *Builder* constructor has a long list of parameters required to deal with each *Concrete Builder*.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Abstract Factory

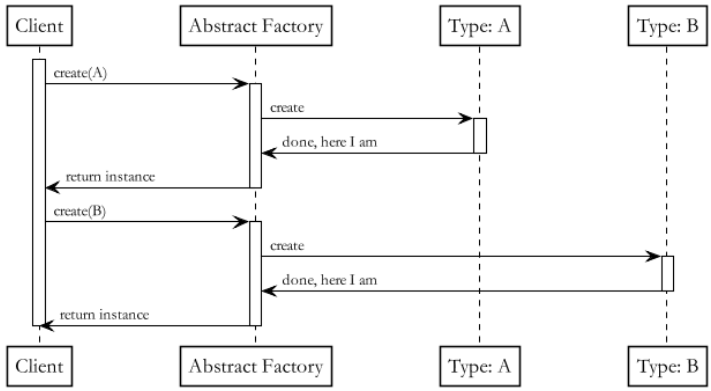


Fig 5.4 - Abstract Factory

The **Abstract Factory** pattern hides the complexity of object creation when slightly different versions of the same object needs to be built.

In other words, the *Abstract Factory* Component internally manages different components that represent different flavors of the final object.

A (very) figurative example is the case of Operating System (OS). The OS can be modeled as an *Abstract Factory*, and it returns an instance of one of the different flavors (different components) it supports (e.g., Linux, Mac, Windows).

In general, this pattern can be used every time we want to support extensions of a conceptually single object.

The Code

An example of python implementation is in the following.

```
# Interface for operations supported by the factory
class AbstractFactory():

    def create_linux(self):
        pass

    def create_win(self):
        pass

# Concrete factory. Managing object creation.
class ConcreteFactoryOS(AbstractFactory):

    def create_linux(self):
        return ConcreteProductLinux()

    def create_win(self):
        return ConcreteProductWin()

# Abstract Linux product
class AbstractProductLinux():

    def interface_linux(self):
        pass

# Concrete linux product
class ConcreteProductLinux(AbstractProductLinux):

    def interface_linux(self):
        print 'running linux'


# Abstract win product
class AbstractProductWin():
```

```
34
35      def interface_win(self):
36          pass
37
38
39  # Concrete win product
40  class ConcreteProductWin(AbstractProductWin):
41
42      def interface_win(self):
43          print 'running win'
44
45  # Factory usage and testing out
46  if __name__ == "__main__":
47      factory = ConcreteFactoryOS()
48      product_linux = factory.create_linux()
49      product_win = factory.create_win()
50      product_linux.interface_linux()
51      product_win.interface_win()
```

Don'ts

Once again: be carefull when adding complexity and layers of abstraction. Indeed, adding a new product is not that scalable since it requires new implementations for each factory.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Factory Method

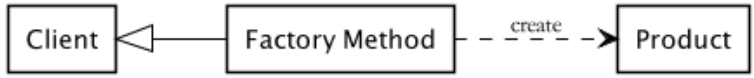


Fig 5.5 - Factory Method

This pattern is similar to the abstract factory, hence often times confused. Guess what? Instead of building a *factory* object, this pattern can be synthesized as a *factory* (actual) method.

The factory method, differently from the abstract factory, instead of dealing with the composition of different sub-objects, it is meant to create an object hiding internal details, whilst being a single concrete product.

The Code

The following snippet shows a simple implementation of the pattern.

```
1  # First Product
2  class ProductA(object):
3      def __init__(self):
4          print ('Building Product A')
5
6  # Second Product
7  class ProductB(object):
8      def __init__(self):
9          print ('Building Product B')
10
```

```
# Factory Method
def factory_method(product_type):
    if product_type == 'PA':
        return ProductA()
    elif product_type == 'PB':
        return ProductB()
    else:
        raise ValueError('Cannot find: {}'.format(p\
roduct_type))


# Client: testing out
if __name__ == '__main__':
    for product_type in ('PA', 'PB'):
        product = factory_method(product_type)
        print(str(product))
```

Don'ts

The recommendation follows the same guideline: do not opt for factories when object abstraction is not needed.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you

tomorrow!

Structural

This Section walks through the principal *Structural* patterns. I like to think about this category as a big puzzle. You have interfaces and code which are already in place. But you still have to make all the components interact and work in the best possible way. The following patterns helps in achieving it.

Class	Patterns
Structural	Adapter
	Decorator
	Facade
	Composite

Adapter

A piece of the puzzle.

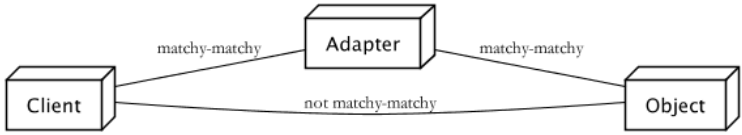


Fig 5.6 - Adapter

What

The adapter is also known as *wrapper*. It wraps another object, redefining its interface.

How

A new class simply encapsulated the incompatible object, thus providing the desired interface.

In the following, a general implementation of this pattern.

```
# Adapter: our wripping class
class Adapter:

    def __init__(self):
        self._adaptee = Adaptee()

    def request(self):
        self._adaptee.legacy_request()


# Adaptee: existing interface
class Adaptee:

    def legacy_request(self):
        print 'Matchy-matchy now! yay!'


# Client: testing out
if __name__ == "__main__":
    adapter = Adapter()
    adapter.request()
```

When

It provides a simple way for solving compatibility issues between different interfaces. Suppose a *caller* is expecting a different interface from a certain object (*callee*) it can be made compatible by means of the *adapter*. They can be handy for legacy software. It enables reusability for a low price.

Don'ts

Fairly trivial, up to the reader.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Decorator

Some software craft.

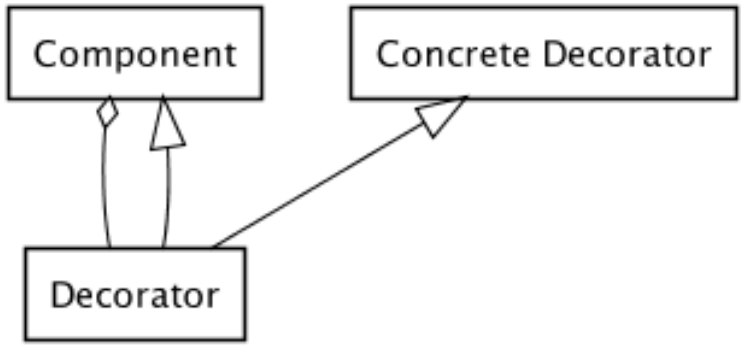


Fig 5.7 - Decorator

What

The Decorator also enable reusability by means of enhancing an object behaviour.

How

Similarly to the previous pattern it wraps the object adding the wanted functionalities.

A simplified example of decorator is shown in the following snippet of code.

```
# Decorator interface
class Decorator:
    def __init__(self, component):
        self._component = component

    def operation(self):
        pass


# Decorator
class ConcreteDecorator(Decorator):
    """
    Add responsibilities to the component.
    """

    def operation(self):
        self._component.operation()
        print 'And some more makeup!'

# Component that needs to be decorated
class Component:

    def operation(self):
        print 'I have some makeup on!'


# Client: testing out
if __name__ == "__main__":
    component = Component()
    decorator = ConcreteDecorator(component)
    decorator.operation()
```

When

It helps in fighting the First Lady Component smell. Thus, it adds functionalities to an object, whilst maintaining Single Responsibility Principle. Indeed, the decorator allows for additional behaviour without impacting the decorated component.

Don’ts

Simple yet powerful, don’t have to add much. But, don’t make the decorator become the new First Lady.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Facade

Putting it all together.

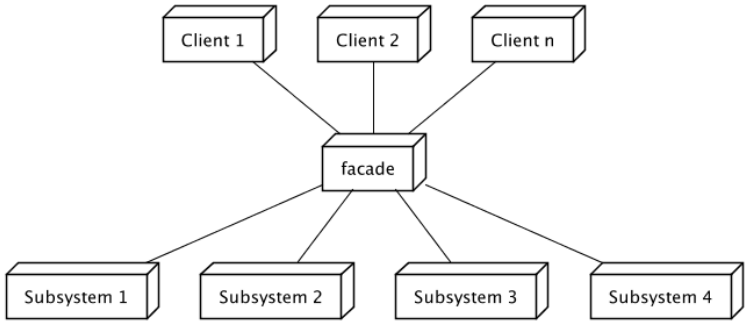


Fig 5.8 - Facade

What

A Facade, can be somehow ideally associated with the Abstract Factory. However, instead of creating an object, it provides a simpler interface for different other more complex interfaces.

How

This pattern provides a brand new higher-level interface in order to make the Subsystems (often independent classes with complex logic) easier to use and interact with.

```
# Facade
class Facade:

    def __init__(self):
        self._subsystem_1 = Subsystem1()
        self._subsystem_2 = Subsystem2()

    def operation(self):
        self._subsystem_1.operation1()
        self._subsystem_2.operation2()

# Subsystem
class Subsystem1:

    def operation1(self):
        print 'Subsystem 1: complex operations'


# Subsystem
class Subsystem2:

    def operation2(self):
        print 'Subsystem 2: complex operations'


# Client
if __name__ == "__main__":
    facade = Facade()
    facade.operation()
```

When

If you are looking at your architecture and it is highly coupled, a facade might help in reducing it.

Don’ts

It is very easy to make a facade that acts as a Single Lady, please avoid it at all costs.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Composite

Uniform is good.

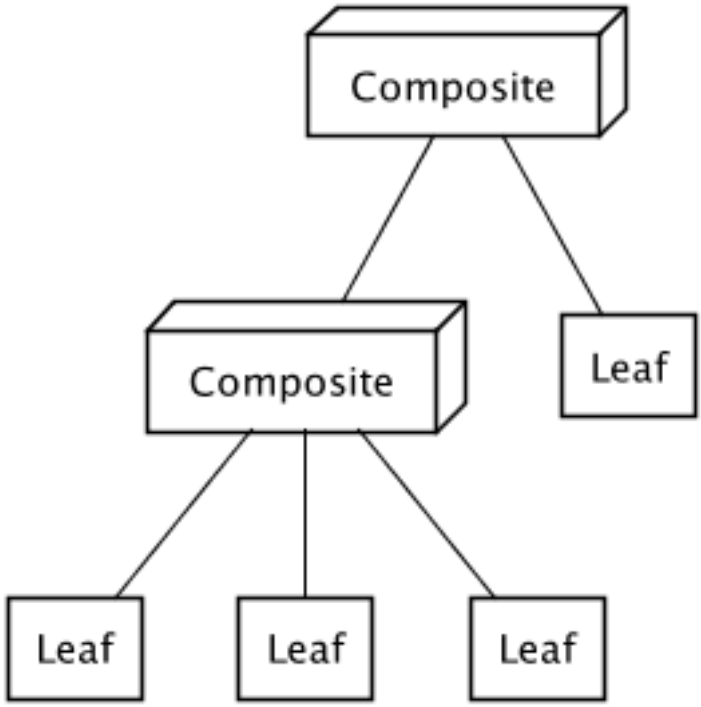


Fig 5.9 - Composite

What

The Composite pattern provides an interface - again. It aims at managing a group of complex objects and single objects exposing similar functionalities in a uniform manner.

How

It composes objects into a tree structure, in such a way that nodes in the tree - unregarding whether they are a leaf (single object) or complex object (i.e. - non leaves) - can be accessed in a similar way, abstracting complexity to the *caller*. In particular, when a method is called on a node, if it is a leaf, the node manages it autonomously. Otherwise, the node calls the method upon its childrens.

```
# Abstract Class
# Defining the interface for all the components in \
the composite
class Component():

    def operation(self):
        pass

# Composite: managing the tree structure
class Composite(Component):

    def __init__(self):
        self._children = set()

    def operation(self):
        print 'I am a Composite!'
        for child in self._children:
            child.operation()

    def add(self, component):
        self._children.add(component)

    def remove(self, component):
```

```
24          self._children.discard(component)
25
26  # Leaf node
27  class Leaf(Component):
28
29      def operation(self):
30          print 'I am a leaf!'
31
32  # Client: testing out
33  if __name__ == "__main__":
34      # Tree structure
35      leaf = Leaf()
36      composite = Composite()
37      composite.add(leaf)
38      composite_root = Composite()
39      leaf_another = Leaf()
40      composite_root.add(composite)
41      composite_root.add(leaf_another)
42
43      # Same operation on the entire tree
44      composite_root.operation()
```

When

This pattern can be used when you have to selectively manage a group of heterogeneous and hierarchical objects as the would ideally the same object. Indeed, this pattern allows for same exploration of the hierarchy, independently from the node type (i.e., leaf and composite).

Don’ts

Take a deeper look at your tree structure. A lot of initialized, whilst not used nodes at the frontier (i.e. leaves), might signal some refactoring required.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Behavioural

Finally, this Section explores Behavioural patterns, e.g., those structures that help in designing common relationships between components.

Class	Patterns
Behavioural	Observer
	Publisher-Subscriber
	Iterator
	Visitor
	State
	Chain of Responsibility

Observer

At a glance.

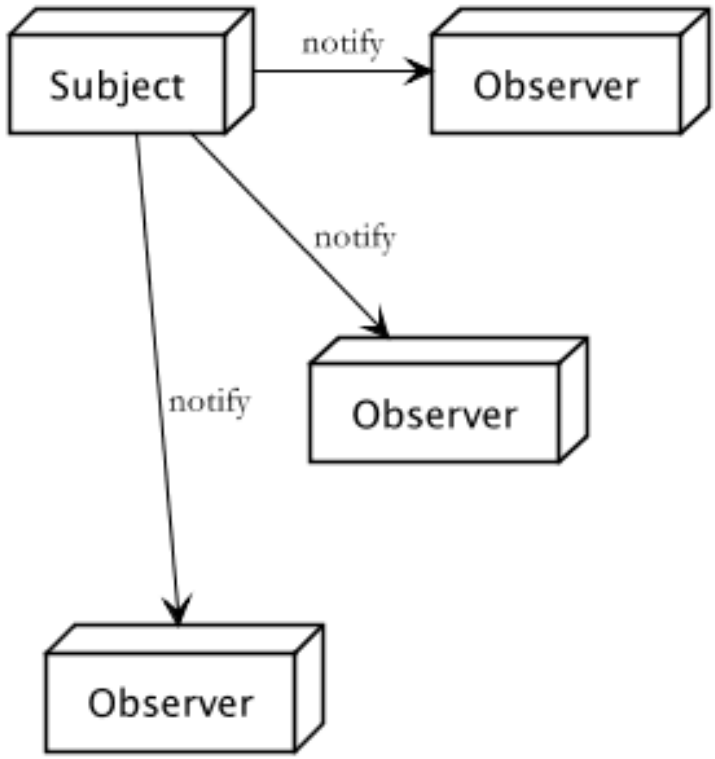


Fig 5.10 - Observer

What

In Operating Systems, a common way of notifying changes happening in the system are *polling* and *interrupts* mechanisms. In the context of higher-level programming, a smarter

way for notifying changes has been ideated: The *Observer* pattern.

How

A component - whose state needs to be notified - stores a list of dependencies. Each and every time a change occurs, it notifies it to its stored list.

```
# The observable subject
class Subject:
    def __init__(self):
        self._observers = set()
        self._state = None

    def subscribe(self, observer):
        observer._subject = self
        self._observers.add(observer)

    def unsubscribe(self, observer):
        observer._subject = None
        self._observers.discard(observer)

    def _notify(self):
        for observer in self._observers:
            observer.update(self._state)

    def set_state(self, arg):
        self._state = arg
        self._notify()

# Interface for the Observer
```

```
class Observer():

    def __init__(self):
        self._subject = None
        self._observer_state = None

    def update(self, arg):
        pass

# Concrete observer
class ConcreteObserver(Observer):

    def update(self, subject_state):
        self._observer_state = subject_state
        print 'Uh oh! The subject changed state to:\
 {}'.format(subject_state)
        # ...

# Testing out
if __name__ == "__main__":
    subject = Subject()
    concrete_observer = ConcreteObserver()
    subject.subscribe(concrete_observer)

    # External changes: testing purposes
    subject.set_state('Ping')
    subject.set_state('Pong')
```

When

When you need different objects to perform - automatically - some functions based on the state of another one (one-to-many).

Don’ts

Once again, keep it simple, evaluate if the conditions above and do not add unnecessary complexity.

Coding Practice of the day

Warm up
TBD

Medium
TBD

Power-up
TBD

See you tomorrow!

Publisher-Subscriber

Please, let me know.

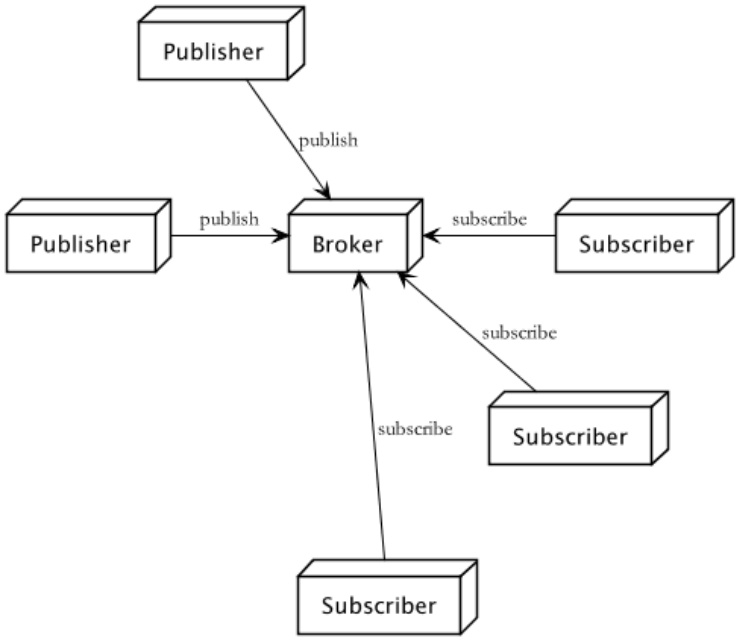


Fig 5.11 - Publisher Subscriber

What

Similarly to the previous one, it enables to monitor state changes.

How

As confusing as it might initially sound, this pattern is very similar to the previous one, but they are not actually the same.

There are two basic components: **publisher** - the entity who's state is monitored, and **subscriber** - the one that is interested in receiving state changes. The main difference is that the dependency between them is abstracted by a third component - often referred to as **broker**, that manages the states update.

```
# Publisher
class Publisher:
    def __init__(self, broker):
        self.state = None
        self._broker = broker

    def set_state(self, arg):
        self._state = arg
        self._broker.publish(arg)


# Subscriber
class Subscriber():
    def __init__(self):
        self._publisher_state = None

    def update(self, state):
        self._publisher_state = state
        print 'Uh oh! The subject changed state to:\
 {}'.format(state)
        # ...

# Broker
class Broker():
    def __init__(self):
        self._subscribers = set()
        self._publishers = set()
```

```
28
29          # Setting up a publisher for testing purpos\
30  es
31          pub = Publisher(self)
32          self._publishers.add(pub)
33
34      # Triggering changes: only for testing purposes
35      def trigger(self):
36          for pub in self._publishers:
37              pub.set_state('Ping')
38
39      def subscribe(self, subscriber):
40          self._subscribers.add(subscriber)
41
42      def unsubscribe(self, subscriber):
43          self._subscribers.discard(subscriber)
44
45      def publish(self, state):
46          for sub in self._subscribers:
47              sub.update(state)
48
49  # Testing out
50  if __name__ == "__main__":
51      # Setting an example
52      broker = Broker()
53      subscriber = Subscriber()
54      broker.subscribe(subscriber)
55
56      # External changes: testing purposes
57      broker.trigger()
```

Be aware that the code provided above is only for showcasing the interactions between the two main components. Some

methods - e.g., *trigger()* - are added only to allow a simple flow of execution.

When

This third-party exchange is helpful in any context where message exchange is required without components (publisher with relative subscribers) being aware of each-others existence.

Don’ts

Don’t overlook the scalability requirements of your solution. The broker might constitute a bottleneck for the entire message exchange.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Iterator

Looking forward.

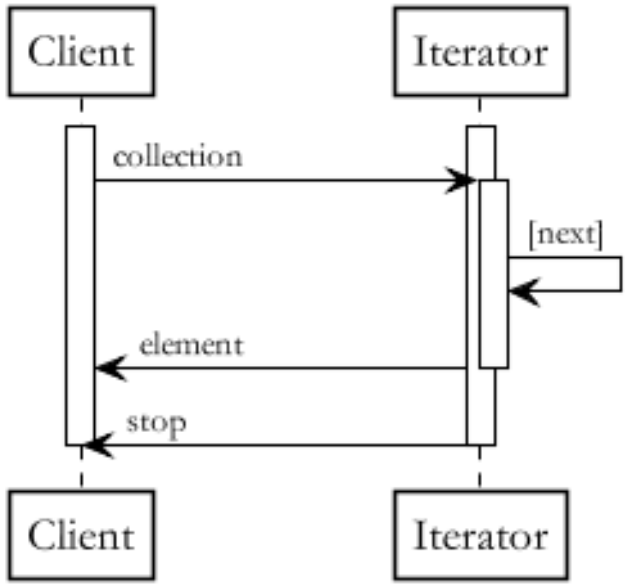


Fig 5.12 - Iterator

What

The iterator allows to navigate elements within an object, abstracting internal management.

How

Commonly, this pattern exposes two method *next()* and *hasNext()* to perform the traversal.

Python implementations normally require an iterable object to implement:

1. **iter**, that returns the instance object.
2. **next**, that will return the next value of the iterable.

A simple implementation of these two method is presented in the following snippet.

```
# Our collection of elements
class MyCollection():

    def __init__(self):
        self._data = list()

    def populate(self):
        for el in range(0, 10):
            self._data.append(el)

    def __iter__(self):
        return Iterator(self._data)

# Our iterator
class Iterator():

    def __init__(self, data):
        self._data = data
        self._counter = 0

    def next(self):
        if self._counter == len(self._data):
            raise StopIteration
```

```
24          to_ret = self._data[self._counter]
25          self._counter = self._counter + 1
26          return to_ret
27
28  # Testing out
29  if __name__ == "__main__":
30      collection = MyCollection()
31      collection.populate()
32      for el in collection:
33          print el
```

In the above example, *StopIteration* signals no more elements in the collection.

When

Probably one of the most common applications is for data structures where elements within them can be (often times sequentially) accessed without knowing inner functioning. However, it can be used any time a traversal is needed without introducing changes to current interfaces.

Don'ts

If the collection is small and simple, it might be not really required.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Visitor

I'll be there is a second.

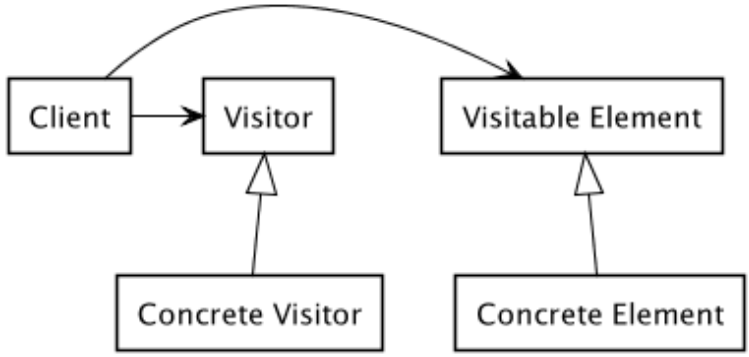


Fig 5.13 - Visitor

What

The Visitor allows to decouple operational logic (i.e., algorithms) that would be - otherwise - scattered throughout different similar objects.

How

A *Visitor* that provides a *visit()* interface that allows to traverse the objects. A *ConcreteVisitor* that implements the actual traversal. A *Visitable* interface that defines an *accept()* method. A *ConcreteVisitable* that given the visitor object implements the accept operation.

```
# Visitable supported operations
class VisitableElement():
    def accept(self, visitor):
        pass

# Concrete element to be visited
class ConcreteElement(VisitableElement):
    def __init__(self):
        self._el = 'Concrete element'

    def accept(self, visitor):
        visitor.visit(self)

# Visitor allowed operations
class Visitor():

    def visit(self, concrete_element_a):
        pass

# Implementing actual visit
class ConcreteVisitor(Visitor):

    def visit(self, concrete_element):
        print 'Visiting {}'.format(concrete_element\
._el)

# Testing out
if __name__ == "__main__":
    concrete_visitor = ConcreteVisitor()
    concrete_element = ConcreteElement()
    concrete_element.accept(concrete_visitor)
```

When

An example of application of the visitor pattern is within data structures for tree traversal (e.g., pre-order, in-order, post-order). It suites fairly well tree-like structures (e.g., syntax parsing), but is not strictly tight to these cases. Visitor pattern is not used only for tree-like structures.

Don'ts

Avoid building visitors around unstable components.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

State

How are you doing?

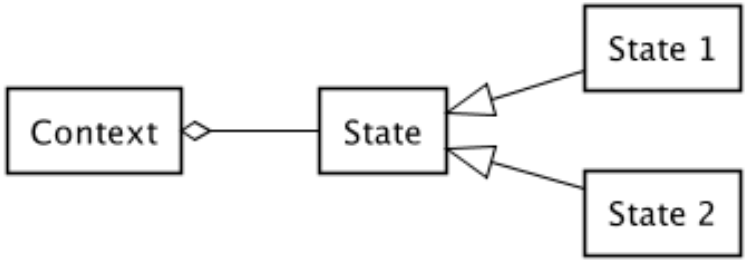


Fig 5.14 - State

What

This pattern enables context-aware objects.

How

It's design is fairly simple: a Context, that represents the external interface, a State abstract class, different State implementations that define the actual states.

In the following a simple implementation of this design pattern.

```
# Context definition
class Context:

    def __init__(self, state):
        self._state = state

    def manage(self):
        self._state.behave()

# Abstract State class
class State():

    def behave(self):
        pass

# Concrete State Implementation
class ConcreteState():

    def behave(self):
        print 'State specific behaviour!'

# Testing out
if __name__ == "__main__":
    state = ConcreteState()
    context = Context(state)
    context.manage()
```

When

The State pattern is helpful each time the behaviour of an object is dependent of its state. In other words, it is applicable when the objects requires changes in behaviour depending on state's changes.

Don’ts

Pay close attention of when you actually use it. The number of states might exponentially grow.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Chain of Responsibility

Micromanaging is never the case.

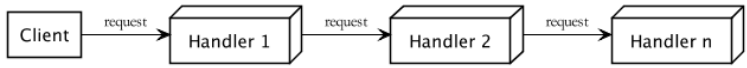


Fig 5.15 - Chain of Responsibility

What

The *Chain of Responsibility* pattern fosters decoupling the sender of request from the receiver.

How

More objects within a pipeline are given a chance to handle an incoming request. In particular, the request is passed sequentially through the pipeline until an object is actually able to handle it.

```
# Handler Interface
class Handler():

    def __init__(self,request=None, successor=None):
        self._successor = successor

    def handle_request(self):
        pass

# Concrete Handler
```

```
class IntegerHandler(Handler):

    def handle_request(self):
        if request.get_type() is int:
            print self.__class__.__name__
        elif self._successor is not None:
            self._successor.handle_request()

# Another Concrete Handler
class StringHandler(Handler):

    def handle_request(self):
        if request.get_type() is str:
            print self.__class__.__name__
        elif self._successor is not None:
            self._successor.handle_request()

# Simple Request object
class Request():

    def __init__(self):
        self._el = 'I am a string'

    def get_type(self):
        return type(self._el)

# Testing out
if __name__ == "__main__":
    request = Request()
    string_handler = StringHandler(request=request)
    int_handler = IntegerHandler(request=request,su\
ccessor=string_handler)
    int_handler.handle_request()
```

When

In cases when you want to simplify the request object and different objects might be added at runtime to actually handle it. It is pretty handy because you can decide which and in which order handlers are added to the chain.

Don'ts

Back to non-functional requirements. Keep an eye on the required performances. Too many handlers (executed sequentially, in worst case skipping up to the very last handler in the chain) might impact code performances. Also bear in mind that debugging these pattern could fairly difficult.

Coding Practice of the day

Warm up

TBD

Medium

TBD

Power-up

TBD

See you tomorrow!

Part 5 - Solutions

The Basics

Basic types

Warm up

Write the Python code that counts the number of character in a string. Print the result.

Solution

```
    def count_char(text):
        return len(text)

    count = count_char('Welcome to the Python World\
')
    print(count)
```

Medium
TBD

Solution

```
1
```

Structure of a Python Program

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Variables

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Python Objects

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Polymorphism

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Conditional Flows

Warm up

Implement a Python program that, given in input a letter *l*, prints if *l* is a vowel or a consonant.

Solution

```
def vowel_or_consonant(l):
    vowels = ['a', 'e', 'i', 'o', 'u']
    if l in vowels:
        print('I am {} and I am a vowel!'.format(l))
    else:
        print('I am {} and I am a consonant!'.forma\
t(l))


if __name__ == "__main__":
    l = 'a'
    vowel_or_consonant(l)
    l = 'g'
    vowel_or_consonant(l)
```

Medium

Implement a Python program that, given in input an object *o*, checks its type (string, integer or none of these).

Hint: study how *isistance()* work.

Solution

```
def check_types(o):
    if isinstance(o, str):
        print ('I am {} and I am a string')
    elif isinstance(o, int):
        print('I am {} and I am an integer')
    else:
        print('I am a mysterious type. hee-hee')

if __name__ == "__main__":
    o = "I am a string"
    check_types(o)

    o = 42
    check_types(o)

    o = list()
    check_types(o)
```

Power-up

TBD

Loops

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Errors

Warm up

TBD

Solution

```
1
```

Medium
TBD

Solution

```
1
```

Documentation

Warm up

TBD

Solution

```
1
```

Medium

TBD

Solution

```
1
```

Algorithms

Data Structures

Array

Warm up

Given an array *A* of integers in input, print it in reverse.

Example

A = [1,2,3,4,5]

Printed Result:

5 4 3 2 1

Solution

```
1    def reverse_array(A):
2        for i in reversed(range(len(A))):
3            print(A[i])
```

Medium

Given an array *A* of integers in input, reverse and print it without using builtin functions.

Example

A = [1,2,3,4,5]

Reverse:

A = [5,4,3,2,1]

Solution

```
def reverse_array(A):
    start = 0
    end = len(A)-1

    while start<end:
        A[start], A[end] = A[end], A[start]
        start+=1
        end-=1

    print(A)
```

Power-up

Given two arrays *A* and *B*, count the occurrences of *B* being a subset of the elements of *A*.

Example 1

A = [1,1,1,1,1]

B = [1,1]

occurrences = 4

Example 2

A = [0,1,1,0,1,0]

B = [0,1]

occurrences = 2

Solution

```
def count_occurrences(A,B):

    # corner case
    if len(B) > len(A):
        return 0

    no_occur = 0
    for i in range(len(A)):
        flag = True
        tmp = i
        for j in range(len(B)):
            if tmp >= len(A) or A[tmp] != B[j]:
                flag = False
                break
```

```
15              tmp += 1
16          if flag:
17              no_occur+=1
18
19      return no_occur
```

Linked-List

Warm up

TBD

Solution

```
1
```

Medium

TBD

Solution

```
1
```

Power-up

TBD

Solution

```
1
```

Stack

Warm up

Given an array *A* of integers in input, print its elements in revers by using a stack.

Example

A = [1,2,3,4,5]

Printed Result

5 4 3 2 1

Solution

```
def reverse_array(A):
    # using list as stack
    stack = []
    for el in A:
        stack.append(el)
    for i in range(len(stack)):
        print(stack.pop())
```

Medium

TBD

Solution

```
1
```

Power-up

Implement a Stack class and provide the logic of main methods: top(), pop() and put().

Solution

```
    class Stack:
        def __init__(self, elements=[]):
            self._stack = []

            # eventually populating the stack
            if elements:
                for el in elements:
                    self.put(el)


        def top(self):
            if self._stack:
                return self._stack[len(self._stack)\
-1]

            raise Exception('Empty stack')

        def pop(self):
            to_ret = self.top()
            self._stack.remove(self._stack[len(self\
._stack)-1])
            return to_ret

        def put(self, el):
            self._stack.append(el)
```

Queue

Warm up

TBD

Solution

```
1
```

Medium

TBD

Solution

```
1
```

Power-up

TBD

Solution

```
1
```

HashMap

Warm up

TBD

Solution

```
1
```

Medium

TBD

Solution

```
1
```

Power-up

TBD

Solution

```
1
```

Binary Search Tree

Warm up

TBD

Solution

```
1
```

Medium

TBD

Solution

```
1
```

Power-up

TBD

Solution

```
1
```

Design Patterns

Conclusions

About the author

Giuliana Carullo, CCSK certified, has computer science in her DNA and has been programming for more than a decade. She holds a Master Degree in Computer Science and she's been doing research for the last six years, whilst wearing another hat: the project manager.

Giuliana is in love with the intersection point between science and human behavior. She believes that there is more than one way to good, much more to do bad, but she ends up being really opinionated on what good is.

In her spare time, she loves to write and to help others in doing their best at their jobs and with their careers.

More from Giuliana Carullo

More from Giuliana Carullo

Code Reviews 101: The Wisdom of Good Coding

Given her strong background on Software Engineering, Giuliana Carullo shows readers how to perform Code Reviews. What you will get away with from this book is knowledge covering a wide scope of challenges and practices on good coding from code, design and architectural smells to measures, processed and methodologies to perform reviews– the right way. If you want to have some fun, check it out. Code Reviews 101[1]

Technical Leadership: Dreams, success and unicorns.

Given her experience with software engineering teams, Giuliana Carullo presents readers a - very opinionated - view on what makes or breaks a good technical leader.

This book is not yet another book on influencing people and on how to communicate properly. These are massively important skills, and plenty of amazing books have been already written on these topics. It born as a vision – often times really personal – on who a technical leader is, how

[1] https://leanpub.com/codereviews101thewisdomofgoodcoding

he/she acts and what it takes to be a good one. Technical Leadership[2]

[2]https://leanpub.com/technicalleadership

Feedback and Errata

Feedback from readers is always more than welcome and highly valued. Let me know what you think about this book. What you liked? What you disliked? What you would like to read in a future version on the topic?

Even if care is taken to ensure accuracy of this book, some errors can happen.

Anything that can go wrong will go wrong - Murphy

If you find a mistake, a typo, something missing, please report it, so I can improve the book.

References

[Cormen et al.]

[PSF] Python Software foundation. https://docs.python.org/3/

[Python Software Foundation]