Scientific Programming Practical 12

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Computing paradigms

Imperative

Imperative programming specifies programs as sequences of statements that change the program's state, focusing on how a program should operate

- C,Pascal

Declarative

Declarative programming expresses the logic of a computation without defining its control flow, focusing on what the program should accomplish

- SQL, Prolog

Object-Oriented

Object-oriented programming is based on the concept of "objects", which may contain data (attributes) and code (methods)

- Java, Smalltalk

Functional

Functional programming treats computation as the evaluation of mathematical functions, avoiding mutable state

- Haskell, OCaml, ML

Computing paradigms: today

Imperative

Imperative programming specifies programs as sequences of statements that change the program's state, focusing on how a program should operate

- C,Pascal

Declarative

Declarative programming expresses the logic of a computation without defining its control flow, focusing on what the program should accomplish

- SQL, Prolog

Object-Oriented

Object-oriented programming is based on the concept of "objects", which may contain data (attributes) and code (methods)

- Java, Smalltalk

Functional

Functional programming treats computation as the evaluation of mathematical functions, avoiding mutable state

- Haskell, OCaml, ML

Object Oriented Programming (OOP)

In Object Oriented Programming (OOP) objects are **data structures that contain data, which is attributes and functions to work with them**. In OOP programs are made by a set of objects that interact with each other.

Class : (types)
define fields and interface to interact (methods)

Objects :
concrete realization of the class

Object Oriented Programming (OOP)

Within a class method, w**e can refer to that very same instance of the object being created by using a special argument** that is called `self`. **self** is always the first argument of each method.

Object Oriented Programming (OOP)

Within a class method, we **can refer to that very same instance of the object being created by using a special argument** that is called `self`. **self** is always the first argument of each method.

Class definition:

```
class class_name:
        #the initilizer method
        def __init__(self, val1,...,valn):
            self.att1 = val1
            ...
            self.attn = valn

        #definition of a method returning something
        def method1(self, par1,...,parn):
            ...
            return value

        #definition of a method returning None
        def method2(self, par1,...,parn):
            ...
```

Object Oriented Programming (OOP)

Within a class method, we **can refer to that very same instance of the object being created by using a special argument** that is called `self`. **self** is always the first argument of each method.

Class definition:

```
class class_name:
        #the initilizer method
        def __init__(self, val1,...,valn):
            self.att1 = val1
            ...
            self.attn = valn

        #definition of a method returning something
        def method1(self, par1,...,parn):
            ...
            return value

        #definition of a method returning None
        def method2(self, par1,...,parn):
            ...
```

Object instantiation:

```
my_class = class_name(p1,...,pn)
```

Example: Let's define a simple class rectangle with two fields (length and width) and two methods (perimeter and area).

```
import math

class Rectangle:
    def __init__(self, l,w):
        self.length = l
        self.width = w

    def perimeter(self):
        return 2*(self.length + self.width)

    def area(self):
        return self.length * self.width

    def diagonal(self):
        return math.sqrt(self.length**2 + self.width**2)
```

Example: Let's define a simple class rectangle with two fields (length and width) and two methods (perimeter and area).

```
import math

class Rectangle:
    def __init__(self, l,w):
        self.length = l
        self.width = w

    def perimeter(self):
        return 2*(self.length + self.width)

    def area(self):
        return self.length * self.width

    def diagonal(self):
        return math.sqrt(self.length**2 + self.width**2)
```

```
R = Rectangle(5,10)
print(type(R))
R1 = Rectangle(5,10)
print(type(R1))
print("R == R1? {} id R:{} id R1:{}".format(R == R1,
                                             id(R),
                                             id(R1)))
p = R.perimeter()
a = R.area()
d = R.diagonal()
print("\nR:\nLength: {} Width: {}\nPerimeter: {}\nArea:{}".format(R.length,
                                                                    R.width,
                                                                    p,
                                                                    a))
print("R's diagonal: {:.2f}".format(d))
R2 = Rectangle(72,13)
p = R2.perimeter()
a = R2.area()
d = R2.diagonal()
print("\nR2:\nLength: {} Width: {}\nPerimeter: {}\nArea:{}".format(R2.length,
                                                                     R2.width,
                                                                     p,
                                                                     a))
print("R's diagonal: {:.2f}".format(d))
```

Result:

```
<class '__main__.Rectangle'>
<class '__main__.Rectangle'>
R == R1? False id R:140386175240232 id R1:140386175240288

R:
Length: 5 Width: 10
Perimeter: 30
Area:50
R's diagonal: 11.18

R2:
Length: 72 Width: 13
Perimeter: 170
Area:936
R's diagonal: 73.16
```

Encapsulation

Setting methods and attributes as private to avoid unwanted interaction with data.
When defining classes, it is possible to hide some of the details that must be kept private to the object itself and not accessed directly. This can be done by setting **methods** and **attributes** (fields) as **private** to the object (i.e. accessible only internally to the object itself).

This can be done by **proceeding variable names with __length (two underscores)**

```
class MyClass:
    def __init__(self, arg1,arg2, arg3):
        self.a1 = arg1
        self.__a2 = arg2
        self.__a3 = arg3
```

Attribute a1 is publically available
Attribute a2 and a3 are private

Encapsulation

Setting methods and attributes as private to avoid unwanted interaction with data.
When defining classes, it is possible to hide some of the details that must be kept private to the object itself and not accessed directly. This can be done by setting **methods** and **attributes** (fields) as **private** to the object (i.e. accessible only internally to the object itself).

This can be done by **proceeding variable names with __length (two underscores)**

```
class MyClass:
    def __init__(self, arg1,arg2, arg3):
        self.a1 = arg1
        self.__a2 = arg2
        self.__a3 = arg3

M = MyClass(1,2,3)

print(M.a1)
print(M.a2)
```

```
1

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-30-b49350c634aa> in <module>()
      8
      9 print(M.a1)
---> 10 print(M.a2)

AttributeError: 'MyClass' object has no attribute 'a2'
```

Encapsulation

Setting methods and attributes as private to avoid unwanted interaction with data.
When defining classes, it is possible to hide some of the details that must be kept private to the object itself and not accessed directly. This can be done by setting **methods** and **attributes** (fields) as **private** to the object (i.e. accessible only internally to the object itself).

This can be done by **proceeding variable names with __length (two underscores)**

Use getters (and setters):

```
class MyClass:
    def __init__(self, arg1,arg2, arg3):
        self.a1 = arg1
        self.__a2 = arg2
        self.__a3 = arg3

    def getA1(self):
        return self.a1

    def getA2(self):
        return self.__a2

    def getA3(self):
        return self.__a3

M = MyClass(1,2,3)

print(M.a1)
print(M.getA2())
```

```
1
2
```

Special methods

it is possible to redefine some operators by redefining the corresponding **special methods** through a process called **overriding**.

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__eq__(self, other)`	`self = other`
`__lt__(self, other)`	`self < other`
`__len__(self)`	`len(self)`
`__str__(self)`	`str(self)`

https://docs.python.org/3/reference/datamodel.html?#special-method-names

Special methods

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__eq__(self, other)`	`self = other`
`__lt__(self, other)`	`self < other`
`__len__(self)`	`len(self)`
`__str__(self)`	`str(self)`

it is possible to redefine some operators by redefining the corresponding **special methods** through a process called **overriding**.

```
class MyClass:
    def __init__(self):
        self.a1 = ["str1", "str2", "str3"]

    def __add__(self, myint):
        tmp = []
        for i in range(len(self.a1)):
            tmp.append(self.a1[i] + "__" + str(myint))
        return tmp

M = MyClass()

print(M.a1)
print(M + 1234)
```

```
['str1', 'str2', 'str3']
['str1__1234', 'str2__1234', 'str3__1234']
```

Inheritance and overriding

One object can **inherit** the attributes and methods from another object. This establishes a "Is-a" relationship between the two objects. The first object is called **subclass** of the original class. A subclass inherits all the methods and attributes of the **superclass**, but it can also redefine some methods through a process called **overriding**.

```
class MySuperClass:
    ...
    def myMethod(self,...):
        ...

class MySubClass(MySuperClass):
    ...
        def myMethod(self,...):
        ...
```

Inheritance and overriding

```
class Person:

    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
        self.age = age

    def getInfo(self):
        return "{} {} is aged {}".format(self.name,
                                         self.surname,
                                         self.age)
```

Inheritance and overriding

```python
class Person:

    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
        self.age = age

    def getInfo(self):
        return "{} {} is aged {}".format(self.name,
                                        self.surname,
                                        self.age)
```

```python
class Dad(Person):
        children = []

        def addChildren(self, children):
            self.children = children

        def addChild(self,child):
            self.children.append(child)

        def getChildren(self):
            return self.children

        def getInfo(self):
            personalInfo = "{} {} is aged {}".format(self.name,
                                                    self.surname,
                                                    self.age)
            childrInfo = ""
            for son in self.getChildren():
                childrInfo += " - {}'s child is {} {}".format(
                            self.name, son.name, son.surname) +"\n"

            return personalInfo + "\n" + childrInfo
```

Inheritance and overriding

```python
class Person:

    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
        self.age = age

    def getInfo(self):
        return "{} {} is aged {}".format(self.name,
                                         self.surname,
                                         self.age)
```

```python
class Dad(Person):
        children = []

        def addChildren(self, children):
            self.children = children

        def addChild(self,child):
            self.children.append(child)

        def getChildren(self):
            return self.children

        def getInfo(self):
            personalInfo = "{} {} is aged {}".format(self.name,
                                        self.surname,
                                        self.age)
            childrInfo = ""
            for son in self.getChildren():
                childrInfo += " - {}'s child is {} {}".format(
                            self.name, son.name, son.surname) +"\n"

            return personalInfo + "\n" + childrInfo
```

```python
jade = Person("Jade", "Smith",5)
print(jade.getInfo())
john = Person("John", "Smith",4)
tim = Person("Tim", "Smith",1)
dan = Dad("Dan", "Smith", 45)
dan.addChildren([jade,john])
dan.addChild(tim)
print(dan.getInfo())
```

```
Jade Smith is aged 5
Dan Smith is aged 45
 - Dan's child is Jade Smith
 - Dan's child is John Smith
 - Dan's child is Tim Smith
```

Functional programming

Programs as if they were stateless mathematical functions

- map : `map(f, input_list)` applies the function `f` to all the elements of `input_list`;
- filter : `filter(f, input_list)` filters `input_list` based on a function `f` that returns true or false for each of the input elements;
- reduce : `reduce(f, input_list)` applies the function `f` to the first two elements of the input list, then it applies it to the result and to the third element and so on until the end of the list is reached and one value only is returned.

Note that the `reduce` function is part of the `functools` module and needs to be imported with:

```
from functools import reduce
```

These return objects, to use the results we need
to convert them to lists, tuples,...

Functional programming

```python
from functools import reduce



myText = "Testing shows the presence, not the absence of bugs"
words = myText.split()

print("Word sizes: {}".format(list(map(len, words))))
cnt = reduce(int.__add__, list(map(len, words)))

print("Dijkstra's quote has {} characters".format(cnt))
```

```
Word sizes: [7, 5, 3, 9, 3, 3, 7, 2, 4]
Dijkstra's quote has 43 characters
```

Lambda functions (anonymous functions)

Their basic syntax is:

```
lambda input-parameters: expression
```

or

```
myfunct = lambda input-parameters: expression
```

Input parameters are comma separated. The expression is separated from the parameters by a colon.

Lambda functions (anonymous functions)

```
sum_lambda = lambda x, y : x+y
mult_lambda = lambda x,y : x*y
cap_lambda = lambda x : x.capitalize()

print(sum_lambda(10,20))
print(sum_lambda("Hi ", "there!"))
print("\n")
print(mult_lambda(10,20))
print(mult_lambda("Hi! ", 3))
print("\n")
txt = "hi there from luca!"
print(cap_lambda(txt))
print(" ".join(map(cap_lambda, txt.split())))
```

```
30
Hi there!


200
Hi! Hi! Hi!


Hi there from luca!
Hi There From Luca!
```

http://qcbprolab.readthedocs.io/en/latest/practical12.html

Exercises

1. Create a Sequence class that can contain DNA sequences assuming "A,G,C,T" as the only characters allowed. Implement a `complement` method that complements the sequence, a `computeGC` method that returns the GC content (i.e. number of G+C/total length of sequence), redefine the "+" operator (`__add__`) to concatenate the sequence with another sequence in input and the `__str__` so that given a DNA sequence "ACTCG" will print it as:

```
5'-ACTCG-3'
3'-TGAGC-5'
```

Show/Hide Solution

2. Define a 3D point class (`Point3D`) which contains three attributes that are the (x,y,z) coordinates in the 3D space and a string (label). Implement a `computeDistance` method that computes the distance between the point and another point (remember that if $a = (x_a, y_a, z_a)$ and $b = (x_b, y_b, z_b)$, $distance(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2}$.

Given the following points:

```
p = Point3D(0,10,0, "alfa")
p1 = Point3D(10,20,10, "point")
p2 = Point3D(4,9, 10, "other")
p3 = Point3D(8,9,11, "zebra")
p4 = Point3D(0,10,10, "label")
p4 = Point3D(0,10,10, "last")
p5 = Point3D(42,102,10, "fifth")
```