Python and Object Oriented Programming (OOP)

IN104

Natalia Díaz Rodríguez, ENSTA, Institut Polytechnique Paris

Python Paradigms

Python allows to work in several paradigms:

- **Imperative** (procedural, classic) programming
  - Most mainstream languages (including **OOP** languages C#, Visual Basic, C++, and Java) were designed to primarily support it

- **Functional** (a form of declarative) programming: an *expression oriented programming*. Ex. of Python expression oriented functions:
  - `map(function_to_apply, list_of_inputs)`
  - `filter(function_to_apply, list_of_inputs)`
  - `reduce(function_to_apply, list_of_inputs)`
  - `lambda argument: manipulate(argument)`

http://book.pythontips.com/en/latest/map_filter.html
http://book.pythontips.com/en/latest/decorators.html
https://www.python-course.eu/python3_lambda.php

Python Paradigms

Python allows to work in several paradigms:

- **Imperative** (procedural, classic) programming
  - Most mainstream languages (including **OOP** languages C#, Visual Basic, C++, and Java) were designed to primarily support it

- **Functional** (a form of declarative) programming: an *expression oriented programming*. Ex. of Python expression oriented functions:
  - `map(function_to_apply, list_of_inputs)`
  - `filter(function_to_apply, list_of_inputs)`
  - `reduce(function_to_apply, list_of_inputs)`
  - `lambda argument: manipulate(argument)`
  - list comprehension

```
[ expression for item in list if conditional ]
```

This is equivalent to:

```
for item in list:
    if conditional:
        expression
```

Python as an OOP language

Object Oriented Programming Paradigm

Python is an OOP language, and in OOP:

- Programs are made up of **object** and **function** definitions
- Most of the computation: in terms of *operations* on objects.
- Correspondence:
  - Each **object** definition <-> Some real world object /concept
  - **Functions** that operate on that object <-> Ways real-world objects interact

Object Oriented Programming Paradigm

You are building a project with many functions with no obvious connection between some class definition and other methods.

Many functions appear to take always at least one **object** of a particular **type**.

=>This observation motivates the needs for a **method**:

a ***function*** associated with a particular ***class*** (i.e., a user-defined type).

Object Oriented Programming Paradigm

Methods:

- Defined inside a class definition
  - make the relationship between the class and the method explicit.
- The syntax to invoke them: different from calling a function.

OOP Notions: Class, Object and Method

- **Object**: **encapsulates** an entity formed by:
  - A state (its data or attributes)
  - Its functioning (methods)
- **Class**: an object template or generic concept used to define its properties and services.
  - **__init__** method: receives initialization parameters (to be assigned to the class member variables, which are declared with ‘self’)
  - ***Self***:
    - Refers to the object itself
    - The first parameter of a method
    - Differentiates parameter names from member variables
      - E.g. self.breads = breads; (Similar to ***this*** in Java)

OOP Notions: Class, Object and Method

- An **object** = a concrete instance of a **class**
- In Python, everything:
  - is an object (in the sense that it can be assigned to a variable or passed as an argument to a function)
    - Functions, Methods, Modules, Classes, and instances of a class are *first-class objects*
  - can have attributes and methods.

Class and Object Syntax

```
class Animal:
    def __init__(self,age,weight):
        self.age = age
        self.__weight = weight

    def __privateMethod(self):
        print(self.weight)

    def getWeight(self):
        return self.__weight

    def eat(self,kgm):
        self.__weight += kgm
        print("The animal weights",self.__weight,"kg after eating.")
```

Objects and Classes Syntax

- Creating an instance (object) of a class: ***ClassName objectName***
- Executing a function: ***functionName(ArgumentList…)***
- Executing a method: ***objectName.methodName(ArgumentList…)***
- Defining a method: ***def methodName(self, ….):***

```
class Animal:
    def __init__(self,age,weight):
        self.age = age
        self.__weight = weight
```

Inheritance

- The ability to define a new class that is a modified version of an existing class.
- The language feature most often associated with OOP.
- A (sub)class B inherits from (super) class A:
  - when B contains (*inherits*) **all the attributes and methods** of parent class A
- Allows:
  - a progressive specialization of classes
  - a larger code reutilization

Inheritance. Syntax:

```
class Bird(Animal):
    def __str__(self):
        return "I am a bird of "+str(self.getWeight())+" kg."
    def fly(self):
        print("I fly as a bird!")
```

Inheritance:

Python allows multiple inheritance

- *Animal* is the superclass of the rest of classes that inherit from it
- *Platypus* inherits from two classes at the same time
- The method *fly()* has different behaviour depending on the concrete type of object
- Many languages do not allow multiple inheritance

Multiple Inheritance. Syntax:

```
class Platypus1(Mammal, Bird):
    def __str__(self):
        return "I am something rare"
    pass
```

Encapsulation

- One of the main advantages of OOP
- Allows to construct objects with methods and attributes that cannot be called externally:
  - E.g. Internal code, Code we don’t want altered
  - Very useful if we want implementation details hidden from a determined class.
    - E.g. the weight of an animal may be stored in a different country (pounds, not Kg).

Encapsulation

- **Getters** and **setters** allow to keep the UI always the same

(independently of this internal codification -which would be private-)

Encapsulation: Interface Example

```
4  class Animal:
5      def __init__(self,age,weight):
6          self.age = age
7          self.__weight = weight
8
9      def __privateMethod(self):
10         print(self.weight)
11
12     def getWeight(self):
13         return self.__weight
14
15     def eat(self,kgm):
16         self.__weight += kgm
17         print("The animal weights",self.__weight,"kg after eating.")
18
```

Encapsulation: Interfaces

Let's make a Vehicle class. What functions and attributes could it have?

Encapsulation: Interfaces

Let's make a Vehicle class. What functions and attributes could it have?

- *horsepower, nb_seats, nb_doors, color, model, and start(), stop(), drive(), turn()*

Encapsulation: Interfaces

Let's make a Vehicle class. What functions and attributes could it have?

- *horsepower, nb_seats, nb_doors, color, model, and start(), stop(), drive(), turn()*

What if the application is for a traffic jam control system?

Encapsulation: Interfaces

Let's make a Vehicle class. What functions and attributes could it have?

- *horsepower, nb_seat, nb_door, color, model, and start(), stop(), drive(), turn()*

What if the application is for a traffic jam control system?

- It would need a different interface
  - *E.g. location, velocity, direction, update_location(), is_stuck_in_traffic()...*

Public methods

In Python: all methods are public

- Except those starting with double underscore: __

Public methods: QUIZ

What would this call produce?

```
71 print(bear.getWeight())
```

```
11  class Animal:
12      def __init__(self, age, weight):
13          self.age = age
14          self.__weight = weight
15
16      def __privateMethod(self):
17          print(self.weight)
18
19      def getWeight(self):
20          return self.__weight
21
22      def eat(self, kgm):
23          self.__weight += kgm
24          print("The animal weights %f kg after eating" % self.__weight)
25
26  class Bird(Animal):
27      def __str__(self):
28          return "I am a bird of "+str(self.getWeight()) + " kg."
29
30      def fly(self):
31          print("I fly as a bird!")
32
33
34  class Mammal(Animal):
```

Public methods: QUIZ

What would this call produce?

```
71  print(bear.getWeight())
```

```
11  class Animal:
12      def __init__(self, age, weight):
13          self.age = age
14          self.__weight = weight
15
16      def __privateMethod(self):
17          print(self.weight)
18
19      def getWeight(self):
20          return self.__weight
21
22      def eat(self, kgm):
23          self.__weight += kgm
24          print("The animal weights %f kg after eating" % self.__weight)
25
26  class Bird(Animal):
27      def __str__(self):
28          return "I am a bird of "+str(self.getWeight()) + " kg."
29
30      def fly(self):
31          print("I fly as a bird!")
32
33
34  class Mammal(Animal):
```

The weight of the bear

Public methods: QUIZ

And…?

```
72  bear.privateMethod()
```

```
11  class Animal:
12      def __init__(self, age, weight):
13          self.age = age
14          self.__weight = weight
15
16      def __privateMethod(self):
17          print(self.weight)
18
19      def getWeight(self):
20          return self.__weight
21
22      def eat(self, kgm):
23          self.__weight += kgm
24          print("The animal weights %f kg after eating" % self.__weight)
25
26  class Bird(Animal):
27      def __str__(self):
28          return "I am a bird of "+str(self.getWeight()) + " kg."
29
30      def fly(self):
31          print("I fly as a bird!")
32
33
34  class Mammal(Animal):
```

Public methods: QUIZ

And…?

```
72 | bear.privateMethod()
```

Runtime error!

```
11  class Animal:
12      def __init__(self, age, weight):
13          self.age = age
14          self.__weight = weight
15
16      def __privateMethod(self):
17          print(self.weight)
18
19      def getWeight(self):
20          return self.__weight
21
22      def eat(self, kgm):
23          self.__weight += kgm
24          print("The animal weights %f kg after eating" % self.__weight)
25
26  class Bird(Animal):
27      def __str__(self):
28          return "I am a bird of "+str(self.getWeight()) + " kg."
29
30      def fly(self):
31          print("I fly as a bird!")
32
33
34  class Mammal(Animal):
```

Line 24, Column 71

```
The animal weights 0.400000 kg after eating
I fly as a bird!
The animal weights 160.000000 kg after eating
I cannot fly, I am a mammal!
I am a mammal of 160 kg.
I am a bird of 30 kg.
I cannot fly, I am a Bird but ostrichs do not fly!
I cannot fly, I am a mammal!
I am something rare
I fly as a bird!
160
Traceback (most recent call last):
  File "Animal.py", line 85, in <module>
    bear.__privateMethod() # AttributeError: Mammal instance has no attribute '__privateMethod'
AttributeError: Mammal instance has no attribute '__privateMethod'
Natalias-MacBook:IN104 2018-19 Nat nataliadiazrodriguez$
```

Python is a Dynamic & Strongly typed Programming language

- Python is a **dynamically typed** language (We don't have to declare the type of variable while assigning a value to a variable in Python).
  - The property of many languages of being able to execute distinct code depending on the **object** that makes the call
    - Based on the use of **inheritance**
  - Allows to reference objects by the superclass type
    - In run time, the derived class will be called instead.
- Python is also a **strongly typed** language: the interpreter keeps track of all variables types
  - Objects still have a type (but determined at runtime)

When identifying classes and objects is not obvious….

Development plan

Start writing functions that read and write global variables (when necessary).

1. Once you get the program working:
    a. Look for associations between global variables
    b. and the functions that use them.
2. Encapsulate related variables as attributes of an object.
3. Transform the associated functions into methods of the new class.

Practical time! You will:

1. Part 1 (**Individual**): Project 0: Unix/Python/ OOP Tutorial: Learning to use classes in context.
   a. Join the course by creating an account in Gradescope.com and using this access code: MJVW68 (Thanks to Pieter Abbeel & Dan Klein's UC Berkeley CS188)
   b. Submit via Gradescope. Deadline: 1 week from today at 23.59 (check always in Gradescope exact date).
2. Part 2 (in pairs): Creating classes
   a. Implement a program that contains the definition of **two classes**, both subclasses of a main **superclass** (You can also build upon the Animal classes theme or other you like. E.g. *Vehicles*.)
   b. Choose **three attributes** that are common to both classes, **two** that are **specific** to each class, and think where to declare them.
   c. Write min. **2 methods** in each class and execute them in the main program.
   d. In your **private** github repository called **IN104_NameA_SurnameA-NameB_SurnameB** (all team members - Note: this exercise is preferable to do, if you can't be a pair, individually), add as collaborators your team mate(s) and your TA.
   e. Create a folder inside called "OOP" that contains your executable programs (.py files)
   f. Show /Send the link to the program in your repository to your TA

Glossary

- **Object-oriented language**: A language that provides features, such as user-defined classes and method syntax, that facilitate object-oriented programming.
- **Object-oriented programming:** A style of programming in which data and the operations that manipulate it are organized into classes and methods.
- **Method**: A function that is defined inside a class definition and is invoked on instances of that class.
- **Subject:** The object a method is invoked on.
- **Operator overloading:** Changing the behavior of an operator like + so it works with a user defined type.
- **Type-based dispatch:** A programming pattern that checks the type of an operand and invokes different functions for different types.
- **Polymorphic**: Pertaining to a function that can work with more than one type.
- **Information hiding:** The principle that the interface provided by an object should not depend on its implementation, in particular the representation of its attributes.

Glossary

- **Class attribute**: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.
- **Instance attribute:** An attribute associated with an instance of a class.
- **Inheritance**: The ability to define a new class that is a modified version of a previously defined class.
- **Parent class**: The class from which a child class inherits.
- **Child class**: A new class created by inheriting from an existing class; also called a “subclass.”
- **IS-A relationship**: The relationship between a child class and its parent class.
- **HAS-A relationship**: The relationship between two classes where instances of one class contain references to instances of the other.
- **Class diagram**: A diagram that shows the classes in a program and the relationships between them. Great to begin a problem by drawing them!

REFERENCES

- M. CC. Lutz (2011). Programming Python (4.a ed.). EE. UU.: O'Reilly Media.
- M. Lutz (2013). Learning Python (5.a ed.). EE. UU.: O'Reilly Media.
- A. Martelli (2017). Python in a Nutshell (3.a ed.). EE. UU.: O'Reilly Media.
- M. Pilgrim (2011). Dive Into Python 3 (online) APress http://www.diveintopython3.net
- A. Cencerrado Barraqué & D. Masip Rodó. El lenguaje Python. UOC.
- R. Gonzalez Duque. Python para todos(on line).

REFERENCES (II) and Acknowledgements

- Pieter Abbeel & Dan Klein's UC Berkeley CS188 for sharing wonderful resources
- Python Software Foundation. Python Official Webpage. <http://www.python.org>
- A. Downey, J. Elkner, C. Meyers. Think Python: How to Think Like a Computer Scientist: Learning with Python.
- Tutorials to learn Python and OOP: http://web.archive.org/web/20080116080043/http://allendowney.com/ip04/hw10/hw10.html
- Resources to learn Scientific Python: scipy, pandas, numpy: https://github.com/paris-saclay-cds/data-science-workshop-2019
- Jean-Didier Garaud. Coding Practices
- Photography: Sandy Skoglund

Reminder: Course & Team Python Conventions

PEP-8: Defines Python coding practices https://www.python.org/dev/peps/pep-0008

- **Indentation**: 4 spaces (good editors will replace <TAB> by 4 spaces)
- **Variables**: `lower_case_with_underscores`
- **Functions**: `lower_case_with_underscores()`
- **Classes**: `UpperCamelCase`
- **Attributes**: `lower_case_with_underscores`
- **Protected attributes**: `_prefixed_with_1_underscore`
- **Constants**: `ALL_CAPS`
- **Modules**: `lowercase` (single word)

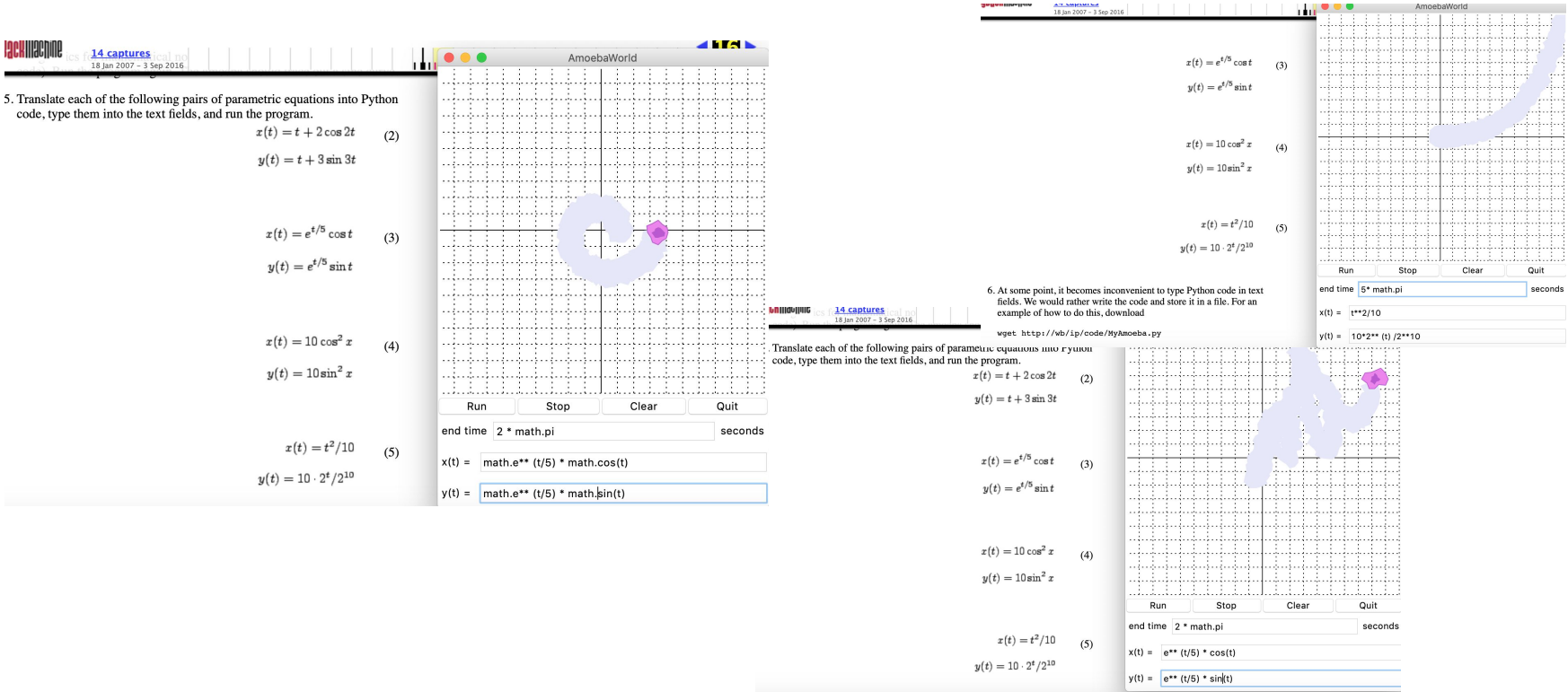
PEP-257: Documentation conventions.

- *Prescribe the function/ method's effect as a command ("Do this", "Return that:").*
- *Docstring should NOT be a "signature" (reiterating the function/method parameters, which can be obtained by instrospection).*

Appendix

[Extra, optional material, not used in this course]

Extra Didactic Tools: Swampy and Amoeba for OOP (yet to be ported to Python 3)



Polymorphism

```
4  class Animal:
5      def __init__(self,age,weight):
6          self.age = age
7          self.__weight = weight
8
9      def __privateMethod(self):
10         print(self.weight)
11
12     def getWeight(self):
13         return self.__weight
14
15     def eat(self,kgm):
16         self.__weight += kgm
17         print("The animal weights",self.__weight,"kg after eating.")
18
19 class Bird(Animal):
20     def __str__(self):
21         return "I am a bird of "+str(self.getWeight())+" kg."
22     def fly(self):
23         print("I fly as a bird!")
24
25 class Mammal(Animal):
26     def __str__(self):
27         return "I am a mammal of "+str(self.getWeight())+" kg."
28     def fly(self):
29         print("I cannot fly, I am a mammal!")
30
31 class Ostrich(Bird,Animal): #Avestruz
32     def fly(self):
33         print("I cannot fly, I am a Bird but ostrichs do not fly!")
```

Polymorphism, Encapsulation & Inheritance in Python classes

```
class Platypus1(Mammal, Bird):
    def __str__(self):
        return "I am something rare"
    pass

class Platypus2(Bird,Mammal):
    def __str__(self):
        return "I am something rare"
    pass


animal1 = Animal(3,0.5)
animal1.eat(0.2)

canary = Bird(1,0.3)
canary.eat(0.1)
canary.fly()

bear = Mammal(10,150)
bear.eat(10)
bear.fly()
print(bear)

ostrich = Ostrich(5,30)
print(ostrich)
ostrich.fly()

platypus = Platypus1(2,3)
platypus.fly()

print(platypus)

platypus = Platypus2(2,3)
platypus.fly()


print(bear.getWeight())
bear.privateMethod()
```

Polymorphism: QUIZ

What lines contain examples of

polymorphism?

a) 47 and 50
b) 36 and 41
c) 55 and 60
d) 63 and 68
e) 60 and 63

```
35 class Platypus1(Mammal, Bird):
36     def __str__(self):
37         return "I am something rare"
38     pass
39
40 class Platypus2(Bird,Mammal):
41     def __str__(self):
42         return "I am something rare"
43     pass
44
45
46 animal1 = Animal(3,0.5)
47 animal1.eat(0.2)
48
49 canary = Bird(1,0.3)
50 canary.eat(0.1)
51 canary.fly()
52
53 bear = Mammal(10,150)
54 bear.eat(10)
55 bear.fly()
56 print(bear)
57
58 ostrich = Ostrich(5,30)
59 print(ostrich)
60 ostrich.fly()
61
62 platypus = Platypus1(2,3)
63 platypus.fly()
64
65 print(platypus)
66
67 platypus = Platypus2(2,3)
68 platypus.fly()
69
70
71 print(bear.getWeight())
72 bear.privateMethod()
```

Python as a functional language

Python as functional language: Advantages

- A function can return a function

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
```

Python as functional language: Advantages

- Saving a function in a variable (to later apply it over arguments)
  - E.g.: f saves a function, and thus, can be called for its execution.

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

Saving functions in a variable: QUIZ

What does line 17 prints?

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

Saving functions in a variable: QUIZ

What does line 18 prints?

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

Saving functions in a variable: QUIZ

What does line 19 prints?

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

Saving functions in a variable: QUIZ

What does line 17 prints? Nada/Nothing/Rien!

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

Saving functions in a variable: QUIZ

What do lines 18 and 19 print?

```
3  def money(country):
4      def spain():
5          print("Euro")
6      def japan():
7          print("Yen")
8      def eeuu():
9          print("dollar")
10
11     functor_money={"es":spain,
12                    "jp":japan,
13                    "us":eeuu}
14
15     return functor_money[country]
16
17 f = money("us")
18 money("us")()
19 f()
```

```
dollar
dollar
```

Lambda functions

- Anonymous
- Not referenced later (single use)
- 1 single line for function + its code
- **lambda** operator: `lambda argument_list: expression`
  - Not followed by parenthesis to indicate parameters
    - these go right after the name of the function, finalized with a colon (**:**)

Lambda functions

```
>>> sum = lambda x, y : x + y
>>> sum(3,4)
7
>>>
```

Equivalent to:

```
>>> def sum(x,y):
...     return x + y
...
>>> sum(3,4)
7
>>>
```

map

Applies 1 function to each element in the list. Returns the new list

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

filter

- Returns the elements of the original list whose (boolean) evaluation of the passed function is true

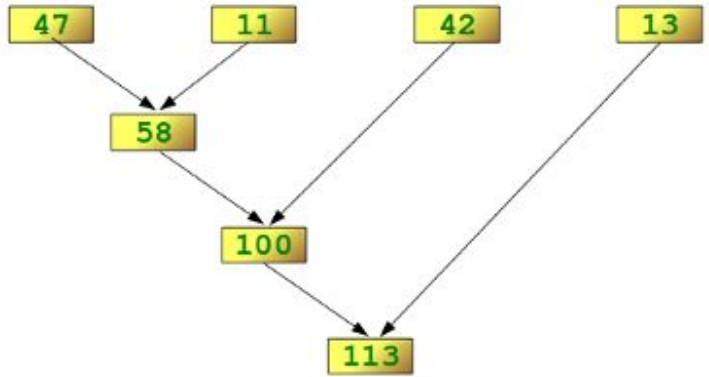
```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

reduce

Applies recursively the passed function to each element of the list until returning a unique result

```
>>> import functools
>>> functools.reduce(lambda x,y: x+y, [47,11,42,13])
113
>>>
```



reduce

Applies recursively the passed function to each element of the list until returning a unique result

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

Equivalent to:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

Python as functional language: Advantages

- **Iterators** (as commonly used in LISP), used in conjunction with lists
- **List Comprehension**: an expression followed by a for loop inside () or []
  - E.g.: for each element of list l, do **expr**:

```
myList = [num * 2 for num in myOtherList]
```

- **`map, filter, reduce`*:*** All limited to 1 expression

(the function they apply over the elements of the list passed as 2nd argument)