Python Advanced

Python and R for Data Science

Data Science and Management

Iterable data

Iterable collections of data

Most data structures in Python are *iterable*, meaning that we can *iterate* over their internal data using a for loop. For instance:

In [524]:

```
L = [0, 1]                                   # a list
for x in L: print("list element:", x) # is iterable

S = {'a', 'b'}                              # a set
for x in S: print("set element:", x) # is iterable

D = {123: 'zero', 456: 'one'}               # a dictionary
for x in D: print("D contains the key", x) # is iterable
```

```
list element: 0
list element: 1
set element: a
set element: b
D contains the key 123
D contains the key 456
```

How to extract all data from an iterable?

When we do not know the actual data type but we know that it is iterable, besides using a for loop, we can extract all the data from anthe iterableusing, e.g., the `list()` function:

In [525]:
```
print(list(L))
print(list(S))
print(list(D))
```

```
[0, 1]
['a', 'b']
[123, 456]
```

Extracting all data from an iterable is expensive because we are creating a new list with all the data. This is not a problem for small data sets, but it can be a problem for large data sets.

``

How to make a piece of data iterable?

1. If we use data types from Python or from popular packages, we may expect that any collection of data is iterable

2. If we define our own data type, we can make it iterable by defining the `__iter__` method. We will return on this after introducing the concept of Python class and object

3. We define a generator function (see next slides)

Generators

The cost of generating values

Suppose we want to implement our own version of `range`:

In [526]:
```
def my_range(n):
    L = []
    i = 0
    while i < n:
        L.append(i)
        i += 1
    return L
```

Such an implementation is correct but is quite inefficient when `n` is a large number e.g, `n=1000000`. Indeed `my_range(n)`:

- needs to iterate `n` times before emitting any kind of result
- needs to store `n` integers

In many cases, we need to generate a series of values but consume them one at a time. This leads us to the concept of *generators*.

Generator

A generator is a function that incrementally produces values and behaves like an iterator, i.e., at each iteration it generates a single distinct value.

To make this possible, the generator function exploits the `yield` statement to return a single value to its caller. When no other values can be generated, the generator `return`s the sentinel value `None` to notify the consumer.

For instance, we can rewrite `my_range`:

In [527]:
```
def my_range(n):
    i = 0
    while i < n:
        yield i
        i += 1
    return None
```

NOTE: the built-in `range` is extremelly flexible and thus its implementation is quite more involved than what we are seeing for `my_range`. Furhtermore, `range` is not technically a generator.

Generators are *lazy*

By design, generators are lazy: they do no eagerly generate the values but wait until we explitly ask for a value using `next()`. For instance:

In [528]:

```
g = my_range(10)
print("Value #0:", next(g))
print("Value #1:", next(g))
print("Value #2:", next(g))
```

```
Value #0: 0
Value #1: 1
Value #2: 2
```

However, Python implicitly calls `next()` over a generator when we use the generator as the target of a for loop:

In [529]:

```
for x in my_range(3):
    print(x)
```

```
0
1
2
```

Killing the spirit of generators

Since generators are iteratable, besides using them in a for loop, we can extract all the data from a generator using, e.g., the `list()` function:

In [530]:
```
list(my_range(5))
```

Out[530]:
```
[0, 1, 2, 3, 4]
```

Again, this may create a performance bottleneck for large data sets. Avoid using `list()` on generators when possible. Use it only for debugging or for small data sets.

More on the `yield` keyword

As anticipated, the `yield` keyword controls the flow of a generator function: the caller

1. it suspends the execution of the generator function and returns a value to
2. it remembers the state of its local variables
3. it does not quit the generator function (as return does)

Lambda Functions

What is a lamba function?

In several cases, we want to quickly define a function that:

1. is extremely short, i.e., 1 line of code
2. is used in our code only once
3. is passed as an argument to other functions (see examples later on!)

A lambda function is thus a convenient way of defining a function. Since we plan to define and use it only once, we do not need to give to the function a name. Hence, a lamba function is said to an anonymous function.

NOTE: you can always use a normal function in the place of a lambda function.

Definition

To *define* a lambda function, the syntax is:

`lambda` (*<ARGS>*) `:` *<CODE>*

where:

- `lambda` is a Python keyword
- *<ARGS>* is the sequence of arguments that your lambda function takes
- *<CODE>* is the line of code processing the arguments (and generating a result)

Let us see an example

Example

This lambda function takes a single argument (`x`) which is incremented and returned:

In [531]:
```
lambda x: x + 1
```

Out[531]:
```
<function __main__.<lambda>(x)>
```

Notice that there is no explicit use of the `return` statement: the return value of the lambda function is what is computed by *<CODE>* (`x+1` in this example).

Our lambda function would be equivalent to the function:

In [532]:
```
def increment(x):
    return x + 1
```

The lambda variant is shorter. However, it does not have a name: since the name helps wrt readability, we use lambda functions only when their task is easy to understand from a quick look. Avoid to use lambda functions when the code logic is cryptic.

When do we use a lambda function?

There are many situations where we call a function that takes another function as an argument. This is common when a function needs an auxiliary function to handle part of its task.

Well-known examples include:

1. Sorting: When a function sorts a collection of values, how do we define the sorting key?
2. Filtering: When a function filters out elements from a collection based on a certain condition, how do we define the filtering criteria?
3. Transformation: When a function transforms values within a collection, how do we specify the transformation to apply to each element?

Python offers several complex functions whose behavior can be tuned by defining the related auxiliary function. Hence, as a programmer, we can solve complex tasks by providing a small and compact function for the auxiliary task.

Use of a lambda function: sorting

For instance, `sorted` takes three arguments:

1. the iterable collection that we want to sort, e.g., a list
2. [optional] `reverse`: a boolean flag indicating whether the sorting should be descending
3. [optional] `key`: a function defining the key to consider when doing the sorting

By default, `sorted` will sort collection by considering all the data within an element. This may not be what we want in several cases. Instead of re-implementing a sorting function from scratch, which is painful and error-prone, we can tune the sorting behavior of `sorted` by passing a function to define the sorting `key`.

Use of a lambda function: sorting (cont'd)

Suppose we have a list of tuples, where the first element in the tuple is the name of a student and the second element of the tuple is his/her matricola:

In [533]:
```
L = [('Amy', '5676'), ('Sheldon', '1234')]
```

If we sort this list with `sorted` we get:

In [534]:
```
sorted(L)
```

Out[534]:
```
[('Amy', '5676'), ('Sheldon', '1234')]
```

This is a bit strange since we may most likely want to sort by matricola (the second field in each tuple). However, by default, `sorted` performs the sorting considering all the data within an element, prioritizing `('Amy', '5676')` since it *starts* with a string `Amy` that is *smaller* than `Sheldon` from `('Sheldon', '1234')`.

Use of a lambda function: sorting (cont'd)

To make a sorting by matricola, we can pass the optional argument `key`: such an argument is a function! We have two options:

1. Define a normal function and then pass it to `sorted`:

In [535]:
```
def extract_key(t):
    return t[1] # we return the matricola

sorted(L, key=extract_key)
```

Out[535]:
```
[('Sheldon', '1234'), ('Amy', '5676')]
```

2. Call `sorted` and define inline a lambda function:

In [536]:
```
sorted(L, key=lambda t: t[1])
```

Out[536]:
```
[('Sheldon', '1234'), ('Amy', '5676')]
```

The second approach is more readable because, when looking at the call to `sorted`, we car immediately see what the function passed as `key` is doing.

Use of a lambda function: sorting (cont'd)

Suppose we want to sort a list of strings based on their lenght, we can easily do it with `sorted` and a lambda function:

In [537]:
```
lst = ["Sparta", "This", "is", "not"]
sorted(lst, key=lambda x: len(x))
```

Out[537]:
```
['is', 'not', 'This', 'Sparta']
```

Use of a lambda function: filtering

Suppose we want to keep students with a name starting with `'A'`. We could

In [538]:
```
def filter_by_a(L):
    new_L = []
    for t in L:
        if t[0][0] == 'A':
            new_L.append(t)
    return new_L

L = [('Amy', '5676'), ('Sheldon', '1234')]
filter_by_a(L)
```

Out[538]:
```
[('Amy', '5676')]
```

Filtering is a very common task and `filter_by_a` is more complex than what we would like to see :(

Use of a lambda function: filtering (cont'd)

Python offers the function `filter` to filter values from an iterable collection. It takes two arguments:

1. A function to be run for each item in the iterable that return `True` when the element must be kept, or `False` when the element should be filtered
2. The iterable collection

For the first argument, we may pass a lambda function. For instance:

In [539]:
```
L = [('Amy', '5676'), ('Sheldon', '1234')]
list(filter(lambda t: t[0][0] == 'A', L))
```

Out[539]:
```
[('Amy', '5676')]
```

NOTE: `filter()` returns an iterable object, hence, see the slides on *iterators* to understand why we need to use `list()` to obtain a printable result from `filter()`.

Use of a lambda function: filtering (cont'd)

If you do not want to define a lambda function, you can still use `filter`:

In [540]:
```
def filter_criteria(t):
    return t[0][0] == 'A'

L = [('Amy', '5676'), ('Sheldon', '1234')]
list(filter(filter_criteria, L))
```

Out[540]:
```
[('Amy', '5676')]
```

This works as expected by is less readable: you have to look at the definition of `filter_criteria` (which may be in another file or way far from the place where you call `filter`).

Use of a lambda function: transformation

The buit-in function `map` applies a function to each element of an iterable collection. It takes two arguments:

1. A function to be applied to each element of the iterable
2. The iterable collection

For instance, suppose we want to transform a list of strings into a list of string lenght, we can easily do it with `map` and a lambda function:

In [541]:
```
lst = ["Bazinga", "Amy", "Sheldon", "Penny"]
list(map(lambda x: len(x), lst))
```

Out[541]:
```
[7, 3, 7, 5]
```

This would be equivalent to:

In [542]:
```
lst2 = []
for s in lst:
    lst2.append(len(s))
print(lst2)
```

```
[7, 3, 7, 5]
```

`with` statement

Handling resources

When we work with resources that need to be properly managed, we need to ensure that the resource is properly released when we are done with it. For instance, when we open a file, we need to close it when we are done with it. For instance:

In [543]:
```
f = open('myfile.txt', 'w') # open a file in write mode ('w')
f.write("LUISS")            # write to the file
f.close()                   # close the file
```

What happens if we do not close the file?

1. The data written to the file may not be saved
2. We may run out of file descriptors: there is a limit of number of files that you can open at the same time

Resource management

In general, several resource have to be properly managed, e.g.,:

1. Files
2. Network connections
3. Database connections
4. Locks
5. ...

To cope with these needs, Python offers the `with` statement.

`with` statement for file handling

Most resources in Python can be convientienly managed using the `with` statement. The `with` statement is used to wrap the execution of a block of code within methods defined by a *context manager*.

For instance:

In [544]:
```
with open('myfile.txt', 'w') as f:
    # some code
    f.write("LUISS")
    # some other code
```

If we write into a file within a `with` statement, the file is automatically closed when the block of code is exited. This is true even if an exception is raised within the block of code.

`with` statement with a database connection

SQLite is a popular relational database that can be used in Python. To connect to a SQLite database, we can use the `sqlite3` package. To keep it internally consistent, we need to close the connection when we are done with it:

In [545]:

```
import sqlite3

# Example of using context manager for database connection
with sqlite3.connect('example.db') as conn:
    cursor = conn.cursor()
    cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, nam
    cursor.execute('INSERT INTO users (name) VALUES (?)', ('Alice',))
    conn.commit()

# The connection is automatically closed after the with block.
```

Objects and Classes

Python is an object-oriented language

Python is an *object-oriented programming (OOP) language*: the programmer can define its own data types, that are known as *classes*.

If $\mathcal{C}$ is a *class*, a value of a *class* $\mathcal{C}$ is called an *object*. In other words, an object is an instance of the class $\mathcal{C}$ and $\mathcal{C}$ can be seen as a blueprint for that object.

Each object contains:

- *instance variables*: i.e., attributes reppresented through variables, used to represent a domain value
- *methods*: i.e., functions through which domain elements can be manipulated

Defining a class

To define a class, we use the `class` keyword. For instance, we can define a class `Person`:

In [546]:
```
class Person:
    # this is an empty class
    pass # this is a placeholder since Python requires at
         # least one statement in the class. pass does nothing
```

This class is not very useful since it does not have any instance variables or methods. We can create an object of this class by calling the class as if it were a function:

In [547]:
```
p = Person() # create an instance of the class
print(p)     # print the instance
print(type(p)) # print the type of the instance
```

```
<__main__.Person object at 0x7ffff0750110>
<class '__main__.Person'>
```

Defining a class with instance variables

To define a class with instance variables, we need to define a special method called `__init__`, often dubbed the *constructor*. This method is called when an object of the class is created. For instance, we can define a class `Person` with two instance variables `name` and `age`:

In [548]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
```

We can now create an object of this class and pass the values for the instance variables `name` and `age`:

In [549]:
```
p = Person('Amy', 'Farrah Fowler')
```

Accessing instance variables

We can access the instance variables of an object using the dot notation:

In [550]:
```
print(p.name)
print(p.surname)
```

```
Amy
Farrah Fowler
```

Defining a class with methods

We can add methods in a class by adding the function definition within the class definition. For instance, we can add a method `greet` to the class `Person`:

In [551]:
```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def greet(self):
        print(f"Hello, my name is {self.name} {self.surname}")
```

Notice that the first argument of a class method is always `self`. This is a reference to the object itself. When we call a method of an object, we do not need to pass the `self` argument: Python does it for us:

In [552]:
```python
p = Person('Amy', 'Farrah Fowler')
p.greet()
```

```
Hello, my name is Amy Farrah Fowler
```

Defining a class with methods (cont'd)

The class methods can take any arbitrary number of arguments. For instance, we can add a method `greet` to the class `Person` that takes an argument `other`:

In [553]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def greet(self, other):
        print(f"Hello, {other.name}, my name is {self.name} {self.surname}")
```

Notice that our `greet` function assumes that `other` has an attribute `name`. If `other` does not have an attribute `name`, the function will raise an exception. Python does not force us to declare the expected type for an argument, which can be quite confusing and error-prone. Nonetheless, this is a design choice of Python that allows for more flexibility.

Defining a class with methods (cont'd)

Methods, including the constructor `__init__`, can have optional arguments. For instance, we can add an optional argument `greeting` to the `greet` method:

In [554]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        self.friends = []

    def greet(self, greeting="Hello"):
        print(f"{greeting}, my name is {self.name} {self.surname}")

p = Person('Amy', 'Farrah Fowler')
p.greet()                   # default greeting
p.greet(greeting="Hi")  # custom greeting
```

```
Hello, my name is Amy Farrah Fowler
Hi, my name is Amy Farrah Fowler
```

Object attributes can be updated

Object attributes can be updated by assigning a new value to them. For instance, we can update the `name` attribute of a `Person` object:

In [555]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def change(self, name, surname):
        self.name = name
        self.surname = surname

p = Person('Amy', 'Farrah Fowler')
p.change('Sheldon', 'Cooper') # change the name and surname
print(p.name)
```

```
Sheldon
```

Object attributes can be updated (cont'd)

In Python, you can update the object attributes even outside the class definition. For instance, we can update the `name` attribute of a `Person` object:

In [556]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

p = Person('Amy', 'Farrah Fowler')
p.name = 'Sheldon'
p.surname = 'Cooper'
print(p.name)
```

```
Sheldon
```

Encapsulation

As seen, in our example, we can update the attribute `name` via:

1. the object method `change`
2. directly accessing the attribute

We should always prefer the first approach. This is because the first approach allows the class to control the update of the attribute. For instance, we can add a check to ensure that the new name is a string:

In [557]:
```
class Person:
    def change(self, name, surname):
        assert type(name) == str, "name must be a string"
        assert type(surname) == str, "name must be a string"
        self.name = name
        self.surname = surname
```

Encapsulation (cont’d)

Most of the time, it is the writer of the class that knows how the attributes should be updated. By using methods to update the attributes, we can ensure that the attributes are updated correctly.

Indeed, we seek to separate:

- the public interface of the class, well described to the end-users via its methods by which we can manipulate objects;
- the inner working of the class, which is private to the class designer and should not be seen (or changed) from "outside of the box".

This principle is known as encapsulation.

Static attributes (class attributes)

In Python, we can define inside a class, outside any method definition, one or more static attributes. Static attributes are shared among all objects of a class. For instance, we can define a static attribute `CITY` for the class `Person`:

In [558]:
```python
class Person:
    CITY = 'Pasadena' # class attribute, shared by all instances

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

p1 = Person('Amy', 'Farrah Fowler')
p2 = Person('Sheldon', 'Cooper')

print(p1.CITY, p2.CITY, Person.CITY)

Person.CITY = 'Rome' # if we change the class attribute
                     # we change it for all instances

print(p1.CITY, p2.CITY, Person.CITY)
```

```
Pasadena Pasadena Pasadena
Rome Rome Rome
```