Python Advanced

Python and R for Data Science

Data Science and Management

LUISS

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 [1532]:

```
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: b
set element: a
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 an iterable using, e.g., the `list()` function:

In [1533]:
```
print(list(L))
print(list(S))
print(list(D))
```

```
[0, 1]
['b', 'a']
[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 [1534]:
```
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 uses a `return` to notify the consumer (NOTE: any function, without an explicit `return`, gets for free a `return None` at its end).

For instance, we can rewrite `my_range`:

In [1535]:

```
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`. Furthermore, `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 [1536]:
```python
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 [1537]:
```python
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 [1538]:
```
list(my_range(5))
```

Out[1538]:
```
[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:

1. it suspends the execution of the generator function and returns a value to the caller
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 a convenient way of defining such 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 [1539]:
```
lambda x: x + 1
```

Out[1539]:
```
<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 [1540]:
```
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 [1541]:
```
L = [('Amy', '5676'), ('Sheldon', '1234')]
```

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

In [1542]:
```
sorted(L)
```

Out[1542]:
```
[('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 [1543]:
```
def extract_key(t):
    return t[1] # we return the matricola

sorted(L, key=extract_key)
```

Out[1543]:
```
[('Sheldon', '1234'), ('Amy', '5676')]
```

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

In [1544]:
```
sorted(L, key=lambda t: t[1])
```

Out[1544]:
```
[('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 [1545]:

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

Out[1545]:

```
['is', 'not', 'This', 'Sparta']
```

Use of a lambda function: filtering

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

In [1546]:
```python
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[1546]:
```
[('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 returns `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 [1547]:
```
L = [('Amy', '5676'), ('Sheldon', '1234')]
list(filter(lambda t: t[0][0] == 'A', L))
```

Out[1547]:
```
[('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 [1548]:
```
def filter_criteria(t):
    return t[0][0] == 'A'

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

Out[1548]:
```
[('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 [1549]:
```
lst = ["Bazinga", "Amy", "Sheldon", "Penny"]
list(map(lambda x: len(x), lst))
```

Out[1549]:
```
[7, 3, 7, 5]
```

This would be equivalent to:

In [1550]:
```
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 [1551]:
```
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 [1552]:

```
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 [1553]:
```python
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 [1554]:
```
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 [1555]:
```
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 0x73e21008f560>
<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 [1556]:
```
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 [1557]:
```
p = Person('Amy', 'Farrah Fowler')
```

Accessing instance variables

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

In [1558]:
```
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 [1559]:
```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 [1560]:
```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 [1561]:
```
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 [1562]:
```
class Person:
    COUNT = 0

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        self.friends = []

    def greet(self, greeting="Hello"):
        Person.COUNT += 1
        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 [1563]:
```
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 [1564]:

```
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 [1565]:
```
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 within a class

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 [1566]:

```
class Person:
    CITY = 'Pasadena' # static class attribute, shared by all instances

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

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

print(p1.city, p2.city)

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

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

```
Pasadena Milan
Pasadena Milan Rome
```

The messy situation of static attributes in Python

A static class attribute in Python behave like a *default* attribute for all the object instances: when they do not have a object attribute with a given name, Python check if there exists a class attribute with the given name. Hence, if we try to set/update the attribute for a specific object, we are actually creating a new instance attribute that shadows the static attribute:

In [1567]:
```
# we access the class attribute using the class name
print(Person.CITY)   # get the class attribute
Person.CITY = 'Rome' # update of the class attribute
                     # which affects all instances

# we access the class attribute using the instance
print(p1.CITY)      # since p1 does not have an object attribute
                    # named CITY, we access the class attribute
p1.CITY = 'Milan'   # we create an object attribute named CITY
                    # which shadows the class attribute
print(p1.CITY)      # we access the object attribute
print(Person.CITY)  # we access the class attribute
                    # It is not affected by the object attribute!
```

```
Pasadena
Rome
Milan
Rome
```

The messy situation of static attributes in Python (cont'd)

In our example, we can see that:

1. By default, no object has an attribute `CITY`. Hence, when we access the attribute `CITY` for the object `p1`, Python looks for a class attribute `CITY` and finds it.
2. When we set the attribute `CITY` for the object `p1`, we are actually creating a new instance attribute `CITY` for `p1`, which shadows the class attribute but only within that object.
3. Hence, we have an object attribute in `p1` called `CITY` and then a static class attribute `CITY` for all the other objects. This is quite confusing and error-prone.

To solve such a confusion: always use the class name to access the static attribute.
For instance, we can access the static attribute `CITY` for the class `Person` using `Person.CITY`. Never use the object name to access the static attribute, otherwise, you may end up creating a new instance attribute that shadows the static attribute, leading to the confusion we have seen.

Making an object iterable

We can make an object iterable by defining the `__iter__` method. This method should return an iterator, an auxiliary object (which most of the time is the original object) implementing the `__next__` method. Such auxiliary object should track the current state of the iterator, extracing as needed the data from our original object.

For instance, we can make a `Counter` class that is iterable:

In [1568]:
```
class Counter:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        # The __iter__ method returns the iterator object itself
        return self

    def __next__(self):
        # The __next__ method defines what to return for each iteration
        if self.current < self.limit:
            self.current += 1
            return self.current
        else:
            # Raising StopIteration tells the loop to stop
            raise StopIteration # sentinel value
```

Making an object iterable (cont'd)

In [1569]:
```
c = Counter(10)
for i in c:
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Make an object pritable

For debugging, it is convenient to `print` an object:

In [1570]:
```
print(p1) # print the instance
```

```
<__main__.Person object at 0x73e1ff561a90>
```

By default, we get a weird string representation of the object. However, we can customize such a behavior by implementing the `__str__` method.

Make an object pritable (cont'd)

In [1571]:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def __str__(self):
        return f"{self.name} {self.surname}"

p3 = Person('Penny', 'Hofstadter')
print(p3) # now get something more meaningful
```

```
Penny Hofstadter
```

Object identity and equivalence

Suppose we have two objects `p1` and `p2` of the class `Person`:

In [1572]:
```
p1 = Person('Amy', 'Farrah Fowler')
p2 = Person('Amy', 'Farrah Fowler')
```

They are different objects but they have the same values for the instance variables.

Are they the same person? This is a philosophical question!

In computer programming, we have typically two distinct concepts:

1. *Object identity*: two objects are the same object if they are stored at the same memory location.
2. *Object equivalence*: two objects are equivalent if they have the same values for their instance variables.

Object identity and equivalence (cont'd)

In Python, we can check if two objects are the same object using the `is` operator:

In [1573]:
```
print(p1 is p2) # False
```

```
False
```

We get a result that is technically correct: they are two distinct objects. However, most of the time, we want to compare the data within the objects, not the objects themselves. Hence, we are more interested about the *equivalence* of the objects, not their *identity*.

Object identity and equivalence (cont'd)

In Python, to check the equivalence, we should use the `==` operator:

In [1574]:
```
print(p1 == p2)
```

```
False
```

We still get that the objects are not equivalent because Python, by default, does not know how to compare two objects of a custom class. Hence, by default, the `==` operator compares the memory location of the objects, not their data, i.e., it checks the object identity, not the object equivalence.

Nonetheless, we can define the `__eq__` method to dictate how two objects of a class should be compared.

Object identity and equivalence (cont'd)

For instance:

In [1575]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def __eq__(self, other):
        return self.name == other.name and self.surname == other.surname

p1 = Person('Amy', 'Farrah Fowler')
p2 = Person('Amy', 'Farrah Fowler')
print(p1 == p2) # True
```

```
True
```

Copy and deep copy of an object

To get a copy of an object, we can use the `copy` package:

In [1576]:

```
import copy

class Person:
    def __init__(self, name, surname, phone_numbers):
        self.name = name
        self.surname = surname
        self.phone_numbers = phone_numbers

    def __eq__(self, other):
        return self.name == other.name \
            and self.surname == other.surname \
            and self.phone_numbers == other.phone_numbers

p1 = Person('Amy', 'Farrah Fowler', ['1234', '5678'])
p3 = copy.copy(p1)
print(p3.name, p3.surname, p3.phone_numbers)
print(p3 is p1)
print(p3 == p1)
```

```
Amy Farrah Fowler ['1234', '5678']
False
True
```

Copy and deep copy of an object (cont'd)

The `copy` function creates a *shallow* copy of the object. This means that the object is copiec but the instance variables are not. Hence, if the instance variables are mutable, the shallow copy will not create a new copy of the instance variables. For instance, in our example:

In [1577]:
```
p3.phone_numbers[0] = '0000' # we change the phone number of p3
print(p1.name, p1.surname, p1.phone_numbers) # p1 is affected!!!
```

```
Amy Farrah Fowler ['0000', '5678']
```

We have update the `phone_numbers` also of `p1`. This is because `p1` and `p2` share the same list of phone numbers since `phone_number` is a `List` and a list is mutable. To avoid such a problem, we can use the `deepcopy` function from the `copy` package:

In [1578]:
```
p1.phone_numbers[0] = '1234' # we fix the phone number of p1
p3 = copy.deepcopy(p1)
p3.phone_numbers[0] = '1111' # we change the phone number of p3
print(p1.name, p1.surname, p1.phone_numbers) # p1 is not affected
```

```
Amy Farrah Fowler ['1234', '5678']
```

Inheritance

Inheritance is a mechanism that allows a class to inherit the attributes and methods of another class. The class that inherits is called a *subclass* (or *child class* or *derived class*) and the class that is inherited is called a *superclass* (or *parent class*). By reapeating such a relantionship, we can thus defined a hierarchy of classes.

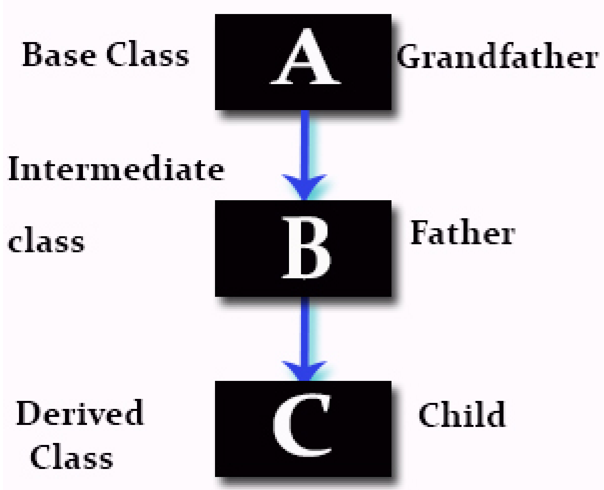


Fig: Multilevel Inheritance

Why do we want inheritance?

Suppose we want to define a class `Student`. If we have already a class `Person`, it makes sense to see `Student` as a specialization of `Person`, i.e., `Student` is derived from `Person` having additional attributes and methods.

In [1579]:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

class Student(Person):
    def __init__(self, name, surname, student_id):
        super().__init__(name, surname) # we need to initialize the parent class
        self.student_id = student_id    # we initialize the attributes
                                        # specific to this class
```

Notice that every instance of class `Student` is also an instance of class `Person`.

is-a relation

In our example, we have built an *is-a* relation between students and persons (because a student *is-a* person), and we have modeled this relation in our Python program by making `Student` a subclass of `Person`. In Python, we can check an *is-a* relation using the built-in `isinstance` function:

In [1580]:
```
s = Student('Sheldon', 'Cooper', '1234')
if isinstance(s, Person):
    print("Student", s.name, s.surname, "with matricola", s.student_id, "is a Per
```

```
Student Sheldon Cooper with matricola 1234 is a Person
```

Notice that, in general, it is not true that a `Person` is a `Student`:

In [1581]:
```
p = Person('Francesco', 'Totti')
if not isinstance(p, Student):
    print("Person", p.name, p.surname, "is not a Student")
```

```
Person Francesco Totti is not a Student
```

Benefit of inheritance: code reuse

Besides establishing a relantionship between our data types, inheritance allows us to reuse the code of the superclass. For instance, we can define a class `Student` that inherits from `Person`:

In [1582]:
```
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}")

class Student(Person):
    def __init__(self, name, surname, student_id):
        super().__init__(name, surname)
        self.student_id = student_id

s = Student('Sheldon', 'Cooper', '1234')
s.greet() # we reuse the method of the parent class
```

```
Hello, my name is Sheldon Cooper
```

A child class can override parent methods

In our example, it could make sense to provide a specialized `greet` for a `Student`:

In [1583]:
```
class Student(Person):
    def __init__(self, name, surname, student_id):
        super().__init__(name, surname)
        self.student_id = student_id

    def greet(self):
        print(f"Hello, my name is {self.name} {self.surname}, and my student ID i

s = Student('Sheldon', 'Cooper', '1234')
s.greet() # we reuse the method of the parent class
```

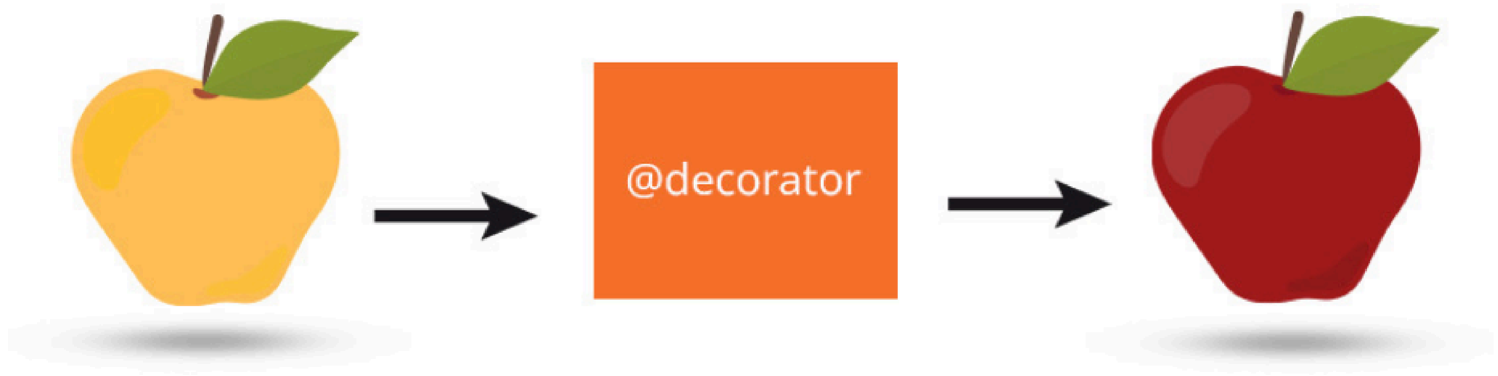
```
Hello, my name is Sheldon Cooper, and my student ID is 1234
```

Broadly speaking, in computer programming, the idea of having the same method (e.g., `greet`) with different behaviors depending on the object type is known as *polymorphism*. In particular, *method overriding* is a form of polymorphism.

Decorators

Function decorators

A function decorator wraps a function, modifying its behaviour. Visually:



Why do need to be aware of decorators?

The community provides a lot of decorators that can be used to improve our code. We get a lot of functionalities for free.

For instance:

1. `@cache` to cache the results of a function
2. `@log` to log the input and output of a function
3. `@timeit` to measure the execution time of a function

Let us see them in action.

Decorators in action: `@cache`

The `@cache` decorator caches the results of a function. For instance:

In [1584]:
```
import functools

@functools.cache
def compute_something(n):
    print("Computing something of", n)
    r = 0
    for i in range(n): r += (200 ** 200) % 1000
    return r # not a very useful computation

print("Run #0:", compute_something(500000))
print("Run #1:", compute_something(500000)) # the result is cached
```

```
Computing something of 500000
Run #0: 0
Run #1: 0
```

The second call to `compute_something(500000)` is much faster than the first call because the result is cached. If you try with a large value of `n` and slow function implementartion, you will see a significant difference in the execution time.

Decorators in action: `@timeit`

In [1585]:

```python
# This is the definition of the decorator
# but you can get it from several packages
def timeit(f):
    def timed(*args, **kw):
        import time
        ts = time.time()
        result = f(*args, **kw)
        print('func:%r args:[%r, %r] took: %2.4f sec' % (f.__name__, args, kw, ti
        return result
    return timed

@timeit
def compute_something(n):
    print("Computing something of", n)
    r = 0
    for i in range(n): r += (200 ** 200) % 1000
    return r # not a very useful computation

compute_something(1)
compute_something(500000)
```

```
Computing something of 1
func:'compute_something' args:[(1,), {}] took: 0.0002 sec
Computing something of 500000
func:'compute_something' args:[(500000,), {}] took: 0.3959 sec
```

```
Out[1585]:    0
```

Parallelism

How does a computer execute code?

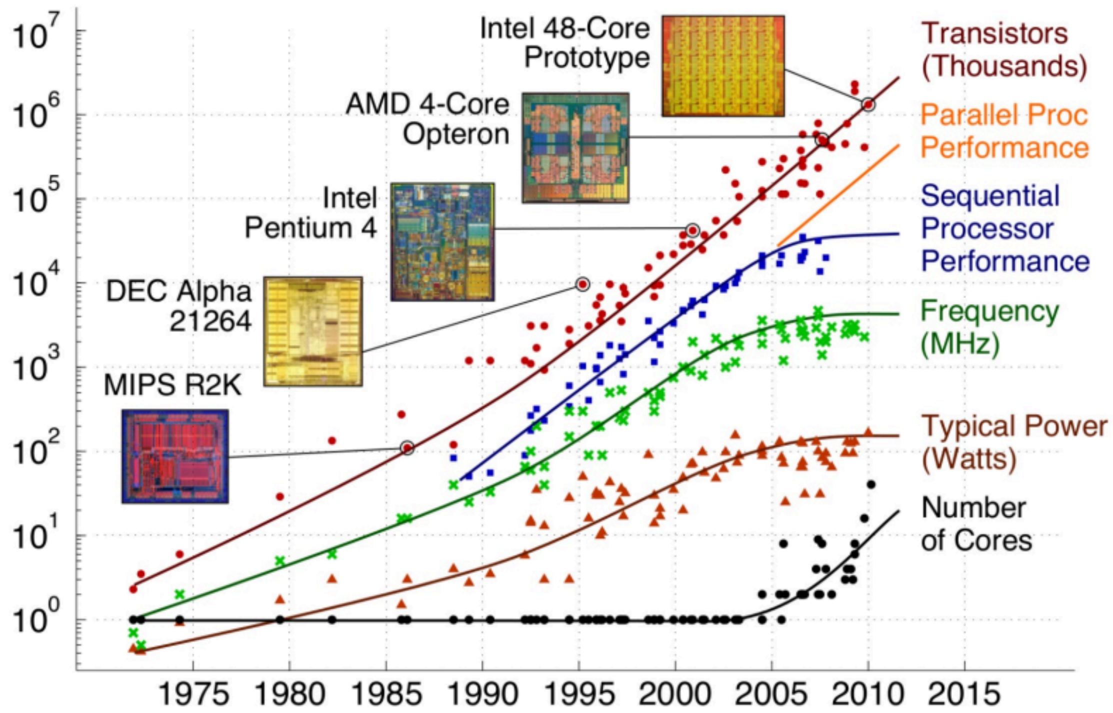
The code of a program is executed by a CPU. A CPU has one or more cores, each of which can execute a single instruction at a time.

How can make our code run faster?

More transistors, more speed (20 years ago)

Moore's Law (1965) states that the number of transistors on a microchip doubles about every two years with a minimal cost increase.

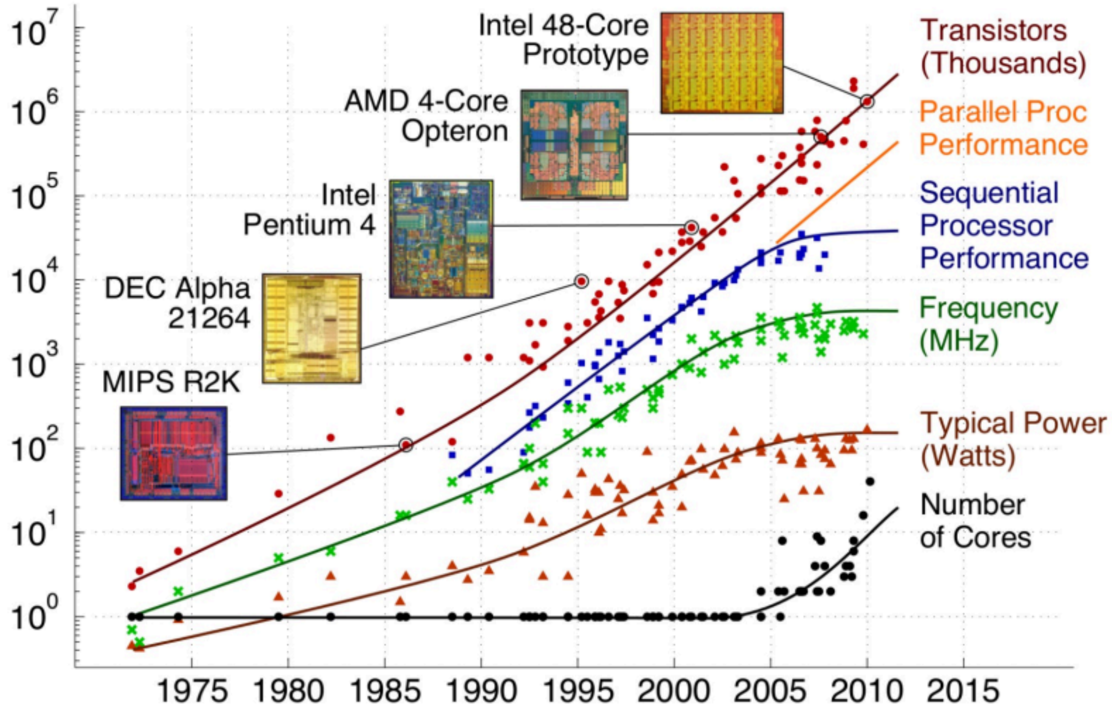
For a long time, more transistors meant more speed.



I:

Free lunch is over: more transistors, more cores

In the past, we could rely on the fact that the CPU speed would double every two years. This is no longer the case. The CPU speed has not increased significantly in the last decade. Instead, the number of cores has increased. The problem is that most of the software is not designed to take advantage of multiple cores.



I:

A Python program likely uses a single core

Most Python code is designed to run on a single core. Up to now, we have we have written code that runs on a single core. Besides technical reasons, the main problem is that writing code that works on different cores is complex and error-prone. For instance:

1. How do we split a larger task into multiple subtasks?
2. How do we synchronize the subtasks?
3. How do we collect the results?

These problems are not easy to solve. After 50 years, there is still no general solution to these problems. In other words, we cannot take a traditional program designed to run over a single core and make it run over multiple cores without rewriting it.

Parallelism in Python

There are several ways to write parallel code in Python. The most common ways are:

1. `threading`: the `threading` module allows us to run multiple *threads* within a single process. Different threads can easily share data. However, due to the Global Interpreter Lock (GIL), Python threads are not suitable for CPU-bound tasks. We will not cover this module.

2. `multiprocessing`: the `multiprocessing` module allows us to run multiple processes Each process has its own memory space and runs in its own memory space. Hence, sharing data is expensive. This is suitable for CPU-bound tasks.

A small cpu-intensive task

Let us consider a syntetic CPU-intensive task: executing 2 times `compute_something(2500000)`

In [1586]:

```
import time

def compute_something(n):
    r = 0
    for i in range(n): r += (200 ** 200) % (1000 + i)
    return r # not a very useful computation

start = time.time()
for i in range(2):
    result = compute_something(2500000)
print("Elapsed time:", time.time() - start, "seconds")
```

```
Elapsed time: 4.117044448852539 seconds
```

`multiprocessing` module: running a function in parallel

In [1587]:

```
import multiprocessing

def compute_something(n):
    r = 0
    for i in range(n): r += (200 ** 200) % (1000 + i)
    return r # not a very useful computation

p1 = multiprocessing.Process(target=compute_something, args=(2500000,))
p2 = multiprocessing.Process(target=compute_something, args=(2500000,))

start = time.time()
p1.start()
p2.start()
results = [p1.join(), p2.join()] # wait for the process to finish
print("Elapsed time:", time.time() - start, "seconds")
```

```
Elapsed time: 2.4804162979125977 seconds
```

On my machine, when using of 2 processes, the execution time is *almost* halved.

`multiprocessing` module: work assignment

What if we have more task than processes?

`multiprocessing.Pool` allows us to create a pool of processes. We can then assign tasks to the pool. The pool will take care of assigning the tasks to the processes.

For instance:

In [1588]:
```
import multiprocessing

def compute_something(n):
    r = 0
    for i in range(n): r += (200 ** 200) % (1000 + i)
    return r # not a very useful computation

pool = multiprocessing.Pool(2)
inputs = [2500000, 2500000, 2500000] # we want to compute the same thing three ti
results = pool.map(compute_something, inputs)
print(results)
```

```
[1561361386212, 1561361386212, 1561361386212]
```

`multiprocessing` module: issues about sharing data

The processes do not share memory. By default, `multiprocessing` takes care of:

1. Transfering the data related to arguments (in our example: `n=2500000`)
2. Transfering the data related to the return value (in our example: `r`)

These operations can be expensive when the data to move around is large. Depending on what we are doing, we may find a way to avoid such a data transfer. For instance, when our function needs to process a file and compute the results, instead of passing the data of the file and get back the data of the processed file, we can just read and write the file from the disk and pass to the function only the file name.

`multiprocessing` module: issues about sharing data (cont'd)

There are several cases when we would like to share data between processes. For instance, we may want to share a large dataset between processes. In such cases, we can use the `multiprocessing.Manager` class to data structures (such as lists) that are shared. However, sharing data is expensive and should be avoided when possible.

For instance:

In [1593]:
```
import multiprocessing

def compute_something(index, l):
    r = 0
    for i in range(l[index]): r += (200 ** 200) % (1000 + i)
    l[index] = r

with multiprocessing.Manager() as manager:
    l = manager.list() # create a shared list
    l.append(2500000)
    l.append(2500000)
    p1 = multiprocessing.Process(target=compute_something, args=(0, l))
    p2 = multiprocessing.Process(target=compute_something, args=(1, l))
    p1.start()
    p2.start()
```

```python
    p1.join()
    p2.join()
    print(l)
```

```
[1561361386212, 1561361386212]
```