

LEARN CODING

ale66

OBJECTS AND CLASSES

RECAP: THE COMPUTATION ARCHITECTURE

- see computer memory as a giant, 1-column spreadsheet
- each cell is defined by 3 features:
 - name;
 - type of content, and
 - actual value

PYTHON CODE IS

- executed a bit like cells
- kept on a separated memory segment
- sequences (strings, lists etc.) are operated in one go by iteration

INTERMEDIATE

methods: pre-cooked functions automatically attached to a variable

```
1 'hello'.upper()  
2  
3 mylist.append('a')  
4  
5 mydict.keys()
```

When a variable is declared, Py. allocates all its *methods* next to it

Large memory occupation

write less, less errors, less surprise results

Python is an object-oriented language.
objects can be understood as extensions of the now-familiar concept of Python variable, which comes with methods attached

PREVIEW

Objects of our new class `temperature`

```
1 temp_milan = temperature('24-nov-2025', 11, 'C')
2
3 temp_seattle = temperature('24-nov-2025', 50, 'F')
```

Custom methods for the class

```
1 # Seattle is actually colder than Milan
2 print(temp_seattle.toC())
3
4 10
```

ADVANCED

Define new types and their specific methods

- a design effort similar to database creation
- will write less, less errors etc.

EXAMPLE: GET CITY TEMPERATURES FROM THE WEB

For city weather, type `int` is adequate, but what if we collect Web data from both metric (SI) and Fahrenheit sources?

Continuously converting values between the two systems errors (and boredom) loom..

THE CONCEPT OF CLASS

CLASS

- a special data type which defines how to create/manage a certain kind of object
- it stores some data that will be shared by all the instances of the class

(ex: how many strings have we created so far?)

- the special `__init__` function create the new object on request
- *instances* are new object/variables created from the class ‘mold’
- no need to dispose of objects

METHODS

- customised functions are defined within the class *block*
- a special `self` argument is used everywhere to remind that we are defining an object
- method `__init__` runs every time we create a new instance

```
1 class temperature:
2     ''' My attempt to work with both Celsius and Fahrenheit temps.
3     '''
4     def __init__(self, date, value = 0, system = 'C'):
5         self.date = TODAY # will fix later
6         self.value = value
7         self.system = system
```

```
1 class temperature:
2     '''My attempt to work with both Celsius and Fahrenheit temps.
3     '''
4     def __init__(self, date, value = 0, system = 'C'):
5         ...
6
7     def toC(self):
8         if self.system == 'C':
9             return self.value
10    else:
11        return # convert F to C: will fix later
```

```
1 class temperature:
2     ...
3
4     def toC(self):
5         if self.system == 'C':
6             return self.value
7         else:
8             return # Convert F to C
9
10    def toF(self):
11        if self.system == 'F':
12            return self.value
13        else:
14            return # Convert C to F: will fix later
```

CREATE INSTANCES

```
1 temp_milan = temperature('22-nov-2024', 11, 'C')
2
3 temp_seattle = temperature('22-nov-2024', 50, 'F')
```

with defaults:

```
1 # acceptable, as C is our default scale
2 temp_rome = temperature(value = 20)
3
4 # this is semantically incorrect
5 temp_guam = temperature(value = 80)
```

THE MEANING OF `self` - 1

```
1  __init__(self, value = 0)
```

here `self` refers to the class we are defining

THE MEANING OF `self` - 2

```
1 my_method(self, value = 0)
```

here `self` refers to the object we are running on

THE MEANING OF `self` - 2 CONT'D

strictly needed in the definition:

```
1 ...
2
3     def my_method(self, value = 0)
4         pass
```

not used in the calls:

```
1 my_object.my_method(value = 27)
```

EXPLORING CLASSES

If we know the class, calls are simple:

```
1 temp_seattle = temperature(value = 50, system = 'F')
2
3 print("It's " + str(temp_seattle.toC()) + ' degrees in Seattle today!')
```

EXPLORING CLASSES CONT'D

Check if a method is there, then call it:

```
1 if hasattr(temp_seattle, 'date'):
2
3     mydate = temp_seattle.date
4
5     print('On ' + str(mydate) + ' it was ' + str(temp_seattle.toC()) + ' in Seattl
```

EXPLORE + APPLY

Sometimes the method to use will only be known at runtime

it cannot be coded in advance, but we can *explore* the object

```
1 # someone defined  
2 temp_seattle = temperature(value = 50, system = 'F')
```

should we forget/don't know, we can query the object to find how it can be used correctly

```
1 print(getattr(temp_seattle, 'system'))  
2  
3 F
```

CLASS ATTRIBUTES

RECAP

As seen above, data attributes essentially are:

- a variable which all instances have a private copy of
- each instance has its own value and can change it

```
1 temp_milan.value
```

CLASS ATTRIBUTES

there is only one variable

all instances will *share the same value*

if an instance changes the value of a class attribute then
everyone gets updated

Applications:

- constants: e.g., there is only one `zero = -273.15 C`
- counters: e.g., create new student objects, but only up to 20

EXERCISE

extend the temperature class to include the Kelvin scale.

```
1 class three_temperatures:  
2     '''Now with three scales!'''  
3  
4     zero = -273.15  
5  
6     def __init__(self, date = date.today(), value = 0, system = 'C'):  
7         self.date = date  
8         self.value = value  
9         self.system = system
```

Use `self.zero` to rebase Celsius degrees to Kelvin

EXERCISE, SOLUTION

extend the temperature class to include the Kelvin scale.

```
1 class three_temperatures:  
2     '''Now with three scales!'''  
3  
4     zero = -273.15  
5     ...  
6  
7     def toK(self):  
8         if self.system == 'C':  
9             return self.value - self.__class__.zero  
10        else:  
11            return self.toC() - self.__class__.zero
```

A MORE GENERAL VERSION

```
1 class three_temperatures:  
2     zero = -273.15  
3     ...  
4  
5     def toK(self):  
6         ...  
7  
8     def generalSciTemp(self, given = self.value,  
9                         scale = self.system):  
10        if scale == 'C':  
11            return given - self.__class__.zero  
12        else:  
13            return ((given -32) * 5/9) - self.__class__.zero
```

```
1 # prints the K equiv. of 100 F: ~311
2 print(getattr(temp_seattle, 'generalSciTemp'))(100))
```

A method can even be *exported* to become a stand-alone function

```
1 # this is a function NAME
2 a_local_function = getattr(temp_seattle, 'generalSciTemp')
3
4 print(a_local_function(100))
5 # prints the same!
```

CLASS COUNTERS

Instances of the same class now can *communicate*

```
1 class student:  
2     '''A student object'''  
3  
4     # class attribute  
5     count = 0  
6  
7     def __init__(self, name, surname = ''):  
8         self.name = name  
9         self.surname = surname  
10        self.__class__.count += 1
```

Class counters are the conceptual base of *distributed computation*

```
1 a = student(name = 'Alice')
2 b = student(name = 'Bob')
3 c = student(name = 'Charlie')
4
5 print(c.__class__.count)
6 # what will it print?
7 # Any difference with, e.g., b.__class__.count
```

PRIVATE DATA AND METHODS

- method/attribute names beginning and ending with `_` are for built-ins: `__init__`
- instead, those starting with `_` but not ending with it remain *private*
- won't be seen outside the class, not even by subclasses

```
1 class student:
2
3     # a regular class attribute
4     count = 0
5
6     # a secret class attribute!
7     __max_capacity = 20
8
9     def __init__(self, name, surname = ''):
10
11         ...
12
13         self.__class__.count += 1
14
15     def __alert(self, count):
16         if count > __max_capacity:
17             print('Class is overbooked!')
```

CLASS INHERITANCE

A class is seldom created from scratch

Often it *extends* a known class, so all setups are inherited

```
1 class geolocated_temp(temperature)
```

All the goodies of **temperature** plus coordinates

```
1 class geolocated_temp(temperature):
2
3     def __init__(self, date, value = 0, system = 'C', place = {'N':"51°31'19.8", 'S':
4         # run the 'inherited' constructor
5         temperature.__init__(self, date, value, system)
6
7         # additionally, set up the place
8         # default: Birkbeck main building
9         self.place = place
10    ...
```

CLASS INHERITANCE, 2

- a new class could inherit from more than one class
- by default, when an object of the new class is created, the `__init__` method of the parent class is called
- example: temperature + float numbers operations to handle scientific temperatures
- the new class can override methods from the parent class
- example: print a float temperature with the date

POLYMORPHISM

A method can be defined in a *parent* class and then re-defined in a *child* class

Current Python uses a complex, dynamic inheritance system: refer to advanced modules

SPECIAL METHODS FOR ALL CLASSES

- attributes define values that are stored for all classes
- methods are automatically attached
- they can be re-defined
- notation: two underscores before and after the name

```
1 __init__ # object constructor
```

```
1 __len__ # defines how to measure objects
```

```
1 __cmp__ # defines how == works for the class
```

```
1 __copy__ # ADVANCED: how to copy a class
```

```
1 __repr__ # defines how to represent the object as a string  
1 >myobject
```

prints by running `myobject.__repr__`

```
1 class student:  
2     ...  
3     def __repr__(self):  
4         return "Hi! I'm " + self.name + ' ' + self.surname + '.'
```

SPECIAL ATTRIBUTES FOR ALL CLASSES

```
1 __doc__
2 __class__
3 __module__
4 __dict__
```

A useful way to explore classes:

```
1 dir(myobject)
2 # returns a list of all the attributes and methods of 'myobject'
```

```
1 __doc__ # documentation string for the class
2
3 __class__ # which is the class of this object?
4
5 __module__ # where was this defined? Numpy? Pandas?
6
7 __dict__ # a dict. of all available functions: the *namespace*
```

```
1 new_temp = temperature(value = 20)
2
3 print(new_temp.__doc__)
4
5 another_temp = new_temp.__class__(value = 30)
6
7 print(new_temp.__module__)
8
9 print(new_temp.__dict__)
```

FINAL CONSIDERATIONS

- as with databases, and unlike Python dictionaries, data are *protected*: only certain functions should access it
- never write the conversion formula, or the 0K again
- don't write, re-use
- a steep learning curve/*cognitive* effort, but then it pays off