LEARN CODING

ale66

github.com/ale66/learn-coding

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

```
'hello'.upper()

mylist.append('a')

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 variables with methods attached are called *objects*

Python is an object-oriented language.

ADVANCED

Define new types and their specific methods

- a design effort similar to SQL Entity-Relationship diagrams
- will write less, less errors etc.

EXAMPLE: GET CITY TEMPERATURES FROM THE WEB

For city wheather, type `int` is adequate, but

what if we collect Web data from both metric and Farenheit 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 object/variables created from the class ‘mould’
- 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

```python
class temperature:
    ''' My attempt to work with both Celsius and Farenheit temps.
    '''
    def __init__(self, date, value = 0, system = 'C'):
        self.date = TODAY # will fix later
        self.value = value
        self.system = system
```

```python
class temperature:
    '''My attempt to work with both Celsius and Farenheit temps.
    '''
    def __init__(self, date, value = 0, system = 'C'):
    ...

    def toC(self):
        if self.system == 'C':
            return self.value
        else:
            return # convert F to C: will fix later
```

```python
class temperature:
    ...

    def toC(self):
        if self.system = 'C':
            return self.value
        else:
            return # Convert F to C

    def toF(self):
        if self.system = 'F':
            return self.value
        else:
            return # Convert C to F: will fix later
```

CREATE INSTANCES

```python
temp_milan = temperature('22-nov-2023', 18, 'C')

temp_seattle = temperature('22-nov-2023', 50, 'F')
```

with defaults:

```
temp_rome = temperature(value = 20)

# this is semantically incorrect...
temp_guam = temperature(value = 80)
```

THE MEANING OF `self` - 1

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

here `self` refers to the class we are defining

THE MEANING OF `self` - 2

```
mymethod(self, value = 0)
```

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

THE MEANING OF `self` - 3

stricly needed in the definitions, not needed in the calls

```
mymethod(self, value = 0)
```

```
myobject.mymethod(value = 27)
```

EXPLORING CLASSES

If we know the class, calls are simple:

```python
temp_sidney = temperature(value = 60, system = 'F')

print("It's " + str(temp_sidney.toC()) + ' degrees in Sidney today!')
```

EXPLORING CLASSES, 2

Check if a method is there, then call it:

```python
if hasattr(temp_sidney, 'date'):

    mydate = temp_sidney.date

    print('On ' + str(mydate) + ' it was ' + str(temp_sidney.toC()) + ' in
```

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

```
# someone defined
temp_sidney = temperature(value = 60, system = 'F')
```

We don't know so we query the object to find how it can be used correctly

```
print(getattr(temp_sidney, 'system'))
# prints 'F'
```

CLASS ATTRIBUTES

DATA ATTRIBUTES

As seen above:

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

```
temp_rome.value
```

CLASS ATTRIBUTES

All instances share the same value

when one inst. changes the value, everyone gets updated

Applications:

- constants (eg., `zero = -273.15 C`)
- counters (eg., create new student objects but only up to 20)

EXERCISE

extend the temperature class to include the Kelvin scale.

```python
class three_temperatures:
    '''Now with three scales!'''

    zero = -273.15

    def __init__(self, date = date.today(), value = 0, system = 'C'):
        self.date = date
        self.value = value
        self.system = system
```

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

EXERCISE, SOLUTION

extend the temperature class to include the Kelvin scale.

```
class three_temperatures:
    '''Now with three scales!'''

    zero = -273.15
    ...

    def toK(self):
        if self.system == 'C':
            return self.value - self.__class__.zero
        else:
            return self.toC() - self.__class__.zero
```

A MORE GENERAL VERSION

```python
class three_temperatures:
    zero = -273.15
    ...

    def toK(self):
        ...

    def generalSciTemp(self, given = self.value,
                             scale = self.system):
        if scale == 'C':
            return given - self.__class__.zero
        else:
            return ((given -32) * 5/9) - self.__class__.zero
```

```python
print(getattr(temp_sidney, 'generalSciTemp')(100))
# prints the K equiv. of 100 F ~311
```

```python
# this is a function NAME
a_local_function = getattr(temp_sidney, 'generalSciTemp')

print(a_local_function(100))
# prints the same!
```

CLASS COUNTERS

Instances of the same class can *communicate* with each other

```python
class student:
    '''A student object.'''

    # class attribute
    count = 0

    def __init__(self, name, surname = ''):
        self.name = name
        self.surname = surname
        self.__class__.count += 1
```

```python
a = student(name = 'Alice')
b = student(name = 'Bob')
c = student(name = 'Charlie')

print(c.__class__.count)
# what will it print?
# Any diference 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

```python
class student:

    count = 0
    # secret class attribute!
    __max_capacity = 25

    def __init__(self, name, surname = ''):
        ...
        self.__class__.count += 1

    def __alert(self, count):
        if count > __max_capacity:
            print('Class is overbooked!')
```

CLASS INHERITANCE

A class is seldom created from scratch

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

```
class geolocated_temp(temperature)
```

All the goodies of `temperature` plus coordinates

```python
class geolocated_temp(temperature):

    def __init__(self, date, value = 0, system = 'C', place = {'N':"51°31'1
        # run the 'inherited' constructor
        temperature.__init__(self, date, value, system)
        # additionally, set up the place
        # default: Birkbeck main building
        self.place = place
    ...
```

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

```
__init__ # object constructor
```

```
__len__ # defines how to meausure objects
```

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

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

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

```
>myobject
```

prints by running `myobject.__repr__`

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

SPECIAL ATTRIBUTES FOR ALL CLASSES

```
__doc__
__class__
__module__
__dict__
```

A useful way to explore classes:

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

```python
__doc__ # documentation string for the class

__class__ # which is the class of this object?

__module__ # where was this defined? Numpy? Pandas?

__dict__ # a dict. of all available functions: the *namespace*
```

```python
new_temp = temperature(value = 20)

print(new_temp.__doc__)

another_temp = new_temp.__class__(value = 30)

print(new_temp.__module__)

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