Lecture 10 – Classes, Objects, OOP

https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

Tomas Jenicek

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics,
Center for Machine Perception
http://cmp.felk.cvut.cz/~jenicto2/
tomas.jenicek@fel.cvut.cz

- **Object-oriented programming language** means it provides features supporting object-oriented programming (OOP)

- OOP main paradigm used in the creation of new software to handle rapidly increasing size and complexity and to make easier to modify and update
- *In Python, everything is an object – everything is an instance of some class*
- In procedural programming the focus is on writing functions or procedures which operate on data

- In **object-oriented programming** the focus is on the creation of objects which contain both **data and functionality together**

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OOP TERMINOLOGY

- **Attribute**: named **data item** that makes up an instance

- **Class**: *compound type* interpretable as a **template** for the objects that are instances of it
- **Class**: is a *prototype for an object* that defines a **set of attributes** that characterize any object of the class.
- The **attributes** (*class variables* and *instance variables*) and **methods** are both accessed via **dot notation**
- **Class variable**: variable **shared by all instances** of the class

  (*class variables are defined within a class but outside any of the class's methods; they are not used as frequently as instance variables are*)

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OOP TERMINOLOGY

- **Data member**: *class* ***variable*** or *instance* ***variable*** that holds data associated with a class and its objects
- **Initializer method**: special method in Python (called **__init__**) that is **invoked automatically** to set a newly created object's attributes to their **initial values**
- **Instance**: object whose type is of some class (*instance and object are used interchangeably*)
- **Instantiate**: procedure necessary to **create an instance** of a class and by running its initializer

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OOP TERMINOLOGY

- **Method**: a function that is defined **inside a class definition** and is **invoked on instances** of that class
- **Object**: a compound data type that is often used to **model a thing or concept** in the real world
- **Object**: bundles together the *data* and the *operations* that are relevant for that kind of data
- **Instance variable**: variable defined **inside a method** that belongs only to the **current instance of a class**
- **Inheritance**: transfer of the characteristics of a class to other classes that are derived from it

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

CLASSES, OBJECTS

```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self):
5          """ Create a new point at the origin """
6          self.x = 0
7          self.y = 0
```

- **EXAMPLE**: *create our own user-defined class: the **Point**. Consider the concept of a mathematical point: in two dimensions, a point is two numbers (coordinates) that are treated as a single object*

- A natural way to represent a point in Python is with two numeric values – *how to group these two values into a compound object*?
- Define a new **class**

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self):
5          """ Create a new point at the origin """
6          self.x = 0
7          self.y = 0
```

- Class definitions are usually near the **beginning** of the program after the import statements, *no need to put every class into its own module*

- Syntax rules for a class definition are the same as for other compound statements
- Header begins with the keyword **class**, followed by the **name** of the class, and ending with a **colon**
- Levels of **indentation** tell us where the class ends

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

CREATING CLASSES

```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self):
5          """ Create a new point at the origin """
6          self.x = 0
7          self.y = 0
```

- The **class** statement creates a new class **definition**
- The class has a documentation string, which can be accessed via **ClassName.__doc__**
- The class suite consists of all the component statements defining class members, data attributes and functions.
- If the *first line after the class header* is a string it is the **docstring** of the class

`__init__` is sometimes called the object's *constructor*, because it is used similarly to the way that constructors are used in other languages, but that is not technically correct – it's better to call it the *initialiser*. There is a different method called `__new__` which is more analogous to a constructor, but it is hardly ever used.

- Every class should have a method with name **__init__**
- This initializer method is automatically called whenever a **new instance is created**
- Initializer is used to **set up the attributes** required within the new instance by giving them their initial state/values
- The **self** parameter (*can have different name but self is the convention*) is automatically set to **reference the newly created object** that needs to be initialized

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html and http://python-textbok.readthedocs.io/en/1.0/Classes.html#

WHAT IS SELF?

- Self is the **first parameter** and we use this variable inside the method bodies – *but we don't appear to pass it, why*?
- Whenever method is called on an object, **the object itself is automatically passed in** as the first parameter giving access the object's properties from inside the object's methods
- In some languages this parameter is implicit (*i.e. it is not visible in the function signature*) and can be accessed with a special keyword
- In Python it is **explicitly exposed** (*very strongly followed convention to name it* **self**)

source http://python-textbok.readthedocs.io/en/1.0/Classes.html#

```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self):
5          """ Create a new point at the origin """
6          self.x = 0
7          self.y = 0
```

```
1  p = Point()          # Instantiate an object of type Point
2  q = Point()          # Make a second point
3
4  print(p.x, p.y, q.x, q.y)  # Each point object has its own x and y
```

```
0 0 0 0
```

- **EXAMPLE**: *The variables **p** and **q** are assigned references to two new **Point** objects*

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

- Think of a class as a **factory for making objects**
- The *class itself is not an instance of a point*, but it contains the **machinery to make point instances**
- Every time the **initializer is called**, the *factory is tasked to make new object*
- As the object is produced, its **initialization method** is executed to get the object properly set up with its factory **default settings**
- The process of making new object and setting it to default settings is called **instantiation**

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

ATTRIBUTES

- Like real world objects, object instances have both **attributes** and **methods**
- Attributes can be modified in an instance using **dot notation**
- Both **modules** and **instances** create their own **namespaces**
- Syntax for accessing attributes (names) is the same

- **EXAMPLE**: *in this case the attribute selected is a data item from an instance (state diagram showing the result of these assignments is below)*

```
>>> p.x = 3
>>> p.y = 4
```

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INITIALIZER


```
1  class Point:
2      """ Point class represents and manipulates x,y coords. """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9  # Other statements outside the class continue below here.
```

```
>>> p = Point(4, 2)
>>> q = Point(6, 3)
>>> r = Point()        # r represents the origin (0, 0)
>>> print(p.x, q.y, r.x)
4 3 0
```

- **EXAMPLE**: *to create a point at position (7, 6) we currently need three lines of code*

```
1  p = Point()
2  p.x = 7
3  p.y = 6
```

- Make class constructor more general by *adding parameters* into the **__init__** method
- The x and y parameters here are **optional** (default values of 0)

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OBJECT METHODS

```
1  class Point:
2      """ Create a new Point, at coordinates x, y """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9      def distance_from_origin(self):
10         """ Compute my distance from the origin """
11         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- Advantage of using a class (*e.g. Point*) rather than a tuple is that **class methods are sensible operations** for points, but may not be appropriate for other data types, e.g. tuples (*e.g. calculate the distance from the origin*)
- Class allows to **group together sensible operations** as well as **data** to apply the methods on
- Each instance of the class has its **own state**
- Method **behaves like a function** but it is invoked on a specific instance

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OBJECT METHODS


```
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.distance_from_origin()
5.0
>>> q = Point(5, 12)
>>> q.x
5
>>> q.y
12
>>> q.distance_from_origin()
13.0
>>> r = Point()
>>> r.x
0
>>> r.y
0
>>> r.distance_from_origin()
0.0
```

```
1  class Point:
2      """ Create a new Point, at coordinates x, y """
3
4      def __init__(self, x=0, y=0):
5          """ Create a new point at x, y """
6          self.x = x
7          self.y = y
8
9      def distance_from_origin(self):
10         """ Compute my distance from the origin """
11         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- First parameter of a method refers to the **instance being manipulated** (the parameter **self**)
- The caller of **distance_from_origin** does not explicitly supply an argument to match the self parameter

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OBJECT METHODS

```
>>> print(p.y)
4
>>> x = p.x
>>> print(x)
3
```

```
1  print("(x={0}, y={1})".format(p.x, p.y))
2  distance_squared_from_origin = p.x * p.x + p.y * p.y
```

- The variable **p** refers to a **Point** object (*containing two attributes referring to the actual numbers*)
- No conflict in the assignment between the variable x (*in the global namespace here*) and the attribute x (*in the namespace belonging to the instance*)
- Purpose of **dot notation** is to *fully qualify* which variable we are referring to unambiguously

- **EXAMPLE**: *the first line outputs (x=3, y=4), the second line calculates the value 25*

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INSTANCES AS ARGUMENTS / PARAMS

```
1  def print_point(pt):
2      print("({0}, {1})".format(pt.x, pt.y))
```

- Pass an **object as an argument** in the usual way
- The variable only holds a reference to an object, therefore **passing object into a function creates an alias** (*both the caller and the called function now have a reference*)
- Function **print_point** takes a point as an argument and formats the output

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INSTANCE TO STRING

```
1  class Point:
2      # ...
3
4      def to_string(self):
5          return "({0}, {1})".format(self.x, self.y)
```

```
>>> p = Point(3, 4)
>>> print(p.to_string())
(3, 4)
```

```
>>> str(p)
'<__main__.Point object at 0x01F9AA10>'
>>> print(p)
'<__main__.Point object at 0x01F9AA10>'
```

- Best approach is to have a **method** so that every instance can produce a string representation of itself
- **TOOLS**: **str** as type converter turns object into a string, **print** function automatically uses this conversion

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INSTANCE TO STRING

```
1      class Point:
2          # ...
3
4          def __str__(self):     # All we have done is renamed the method
5              return "({0}, {1})".format(self.x, self.y)
```

```
>>> str(p)      # Python now uses the __str__ method that we wrote.
(3, 4)
>>> print(p)
(3, 4)
```

- **RECOMMENDATION**: Define the standard method **__str__**
- If method **__str__** is used instead of **to_string**, Python interpreter will use the defined code whenever it needs to convert a Point to a string automatically

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INSTANCES AS RETURN VALUES


```
1  def midpoint(p1, p2):
2      """ Return the midpoint of points p1 and p2 """
3      mx = (p1.x + p2.x)/2
4      my = (p1.y + p2.y)/2
5      return Point(mx, my)
```

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = midpoint(p, q)
>>> r
(4.0, 8.0)
```

- Functions and methods **can return instances**
- **EXAMPLE**: *assume a point object in 2D and aim to find the midpoint halfway between it and some other target point* (function *midpoint*)

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

INSTANCES AS RETURN VALUES


```
1  class Point:
2     # ...
3
4     def halfway(self, target):
5           """ Return the halfway point between myself and the target """
6           mx = (self.x + target.x)/2
7           my = (self.y + target.y)/2
8           return Point(mx, my)
```

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = p.halfway(q)
>>> r
(4.0, 8.0)
```

- **EXAMPLE:** *Implement the midpoint function as method halfway instead (method is identical to the function on the previous slide)*
- As function calls are **composable**, method calls and object instantiation are also **composable**, leading to this alternative that uses **no variables**:

```
>>> print(Point(3, 4).halfway(Point(5, 12)))
(4.0, 8.0)
```

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

OOP PERSPECTIVE

OOP is about changing the perspective

- Syntax for a function call: **function_name(variable)**
  function is the one who executes on the variable

- Syntax in OOP: **object_name.function_name()**
  object is the one who executes its method on given data / attribute

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

EXAMPLE – CLASS VARIABLE


```
>>> class Test(object):
...     i = 3
...
>>> Test.i
3
```

```
>>> t = Test()
>>> t.i     # static variable accessed via instance
3
>>> t.i = 5 # but if we assign to the instance ...
>>> Test.i  # we have not changed the static variable
3
>>> t.i     # we have overwritten Test.i on t by creating a new attribute t.i
5
>>> Test.i = 6 # to change the static variable we do it by assigning to the class
>>> t.i
5
>>> Test.i
6
>>> u = Test()
>>> u.i
6           # changes to t do not affect new instances of Test
```

source https://stackoverflow.com/questions/68645/static-class-variables-in-python

EXAMPLE – CLASS VARIABLE


```
In[2]: class Test:
   ...:     result = 1
   ...:     print(id(result))
   ...:
   ...:     def add(self):
   ...:         self.result += 1
   ...:
   ...:
   ...: def test_a():
   ...:     test = Test()
   ...:     print(Test.result)
   ...:     test.add()
   ...:     print(id(test.result))
   ...:     print(test.result)
   ...:
   ...:
   ...: def test_b():
   ...:     test = Test()
   ...:     print(Test.result)
   ...:     test.add()
   ...:     print(id(test.result))
   ...:     print(test.result)
   ...:
   ...:
   ...: test_a()
   ...: test_b()
   ...: print(id(Test.result))
   ...:
4555148736
1
4555148768
2
1
4555148768
2
4555148736
```

```
In[2]: class Test:
   ...:     result = []
   ...:     print(id(result))
   ...:
   ...:     def add(self):
   ...:         self.result.append('hit')
   ...:
   ...:
   ...: def test_a():
   ...:     test = Test()
   ...:     test.add()
   ...:     print(id(test.result))
   ...:     print(test.result)
   ...:
   ...:
   ...: def test_b():
   ...:     test = Test()
   ...:     test.add()
   ...:     print(id(test.result))
   ...:     print(test.result)
   ...:
   ...:
   ...: test_a()
   ...: test_b()
   ...: print(id(Test.result))
   ...:
4523688008
4523688008
['hit']
4523688008
['hit', 'hit']
4523688008
```

source https://stackoverflow.com/questions/68645/static-class-variables-in-python

DECORATORS


```
In[2]: def my_decorator(some_function):
   ...:     def wrapper():
   ...:         print("Something is happening before some_function() is called.")
   ...:         some_function()
   ...:         print("Something is happening after some_function() is called.")
   ...:
   ...:     return wrapper
   ...:
   ...:
   ...: def just_some_function():
   ...:     print("Wheee!")
   ...:
   ...:
   ...: just_some_function = my_decorator(just_some_function)
   ...: just_some_function()
   ...:
Something is happening before some_function() is called.
Wheee!
Something is happening after some_function() is called.
```

```
In[2]: def my_decorator(some_function):
   ...:     def wrapper():
   ...:         print("Something is happening before some_function() is called.")
   ...:         some_function()
   ...:         print("Something is happening after some_function() is called.")
   ...:
   ...:     return wrapper
   ...:
   ...:
   ...: @my_decorator
   ...: def just_some_function():
   ...:     print("Wheee!")
   ...:
   ...:
   ...: # just_some_function = my_decorator(just_some_function)
   ...: just_some_function()
   ...:
Something is happening before some_function() is called.
Wheee!
Something is happening after some_function() is called.
```

@classmethod

- In the same way class attributes are defined, which are shared between all instances of a class, class methods are defined using @classmethod **decorator** for ordinary method

- Class method still has its calling object as the first parameter, but by convention it is **cls** instead of **self**
- If class method is called from an instance, this parameter will contain the instance object, but if it is called from the class it will contain the class object
- Naming the parameter **cls** serves as reminder that it is not guaranteed to have any instance attributes

What are class methods good for?

- For tasks associated with a class utilizing constants and other class attributes without the need to create any class instances
- **EXAMPLE**: *when we write classes to group related constants together with functions which act on them – no need to instantiate these classes at all*

EXAMPLE – INSTANCE METHODS

```
class Inst:

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

    def introduce(self):
        print("Hello, I am %s, and my name is " %(self, self.name))
```

```
myinst = Inst("Test Instance")
otherinst = Inst("An other instance")
myinst.introduce()
# outputs: Hello, I am <Inst object at x>, and my name is Test Instance
otherinst.introduce()
# outputs: Hello, I am <Inst object at y>, and my name is An other instance
```

SOURCE https://stackoverflow.com/questions/17134653/difference-between-class-and-instance-methods

EXAMPLE – CLASS METHODS

```
class Cls:

    @classmethod
    def introduce(cls):
        print("Hello, I am %s!" %cls)
```

```
 Cls.introduce() # same as Cls.introduce(Cls)
 # outputs: Hello, I am <class 'Cls'>
```

Notice that again `Cls` is passed hiddenly, so we could also say `Cls.introduce(Inst)` and get output `"Hello, I am <class 'Inst'>`. This is particularly useful when we're inheriting a class from `Cls`:

```
class SubCls(Cls):
    pass

SubCls.introduce()
# outputs: Hello, I am <class 'SubCls'>
```

SOURCE https://stackoverflow.com/questions/17134653/difference-between-class-and-instance-methods

STATICS METHODS

@staticmethod

- Static method does not have the calling object passed into it as the first parameter
- Static method does not have access to the rest of the class or instance
- Static method is most commonly called from class objects (*like class methods*)

EXAMPLE – STATIC METHODS


```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

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

    def fullname(self):  # instance method
        # instance object accessible through self
        return "%s %s" % (self.name, self.surname)

    @classmethod
    def allowed_titles_starting_with(cls, startswith):  # class method
        # class or instance object accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]

    @staticmethod
    def allowed_titles_ending_with(endswith):  # static method
        # no parameter for class or instance object
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]
```

```
In[3]: jane = Person("Jane", "Smith")
In[4]: print(jane.fullname())
Jane Smith
In[5]: print(jane.allowed_titles_starting_with("M"))
['Mr', 'Mrs', 'Ms']
In[6]: print(Person.allowed_titles_starting_with("M"))
['Mr', 'Mrs', 'Ms']
In[7]: print(jane.allowed_titles_ending_with("s"))
['Mrs', 'Ms']
In[8]: print(Person.allowed_titles_ending_with("s"))
['Mrs', 'Ms']
```

@property

- Method to generate a property of an object **dynamically** (*e.g. calculating it from the object's other properties*)
- Use a method to **access a single attribute and return it**
- Use a different method to **update the value of the attribute** instead of accessing it directly
- These methods are called **getters** and **setters**, because they "**get**" and "**set**" the values of attributes, respectively

EXAMPLE – PROPERTY

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

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

    @fullname.setter
    def fullname(self, value):
        # this is much more complicated in real life
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname

    @fullname.deleter
    def fullname(self):
        del self.name
        del self.surname
```

IMPORTANT OBJECT PROPERTIES

- `__init__`: the initialisation method of an object, which is called when the object is created.
- `__str__`: the string representation method of an object, which is called when you use the `str` function to convert that object to a string.
- `__class__`: an attribute which stores the the class (or type) of an object – this is what is returned when you use the `type` function on the object.
- `__eq__`: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator `==` to check if two objects are equal.
- `__add__` is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithemtic operations – numbers have all of these methods defined, but other objects may only have a subset.
- `__iter__`: a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the `iter` function on the object.
- `__len__`: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the `len` function of an object.
- `__dict__`: a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. `__dict__` does not include any methods, class attributes or special default attributes like `__class__`.

EXAMPLE – OBJECT PROPERTIES


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

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

    def __gt__(self, other):  # is self > other?
        if self.surname == other.surname:
            return self.name > other.name
        return self.surname > other.surname

    # now we can define all the other methods in terms of the first two

    def __ne__(self, other):  # does self != other?
        return not self == other  # this calls self.__eq__(other)

    def __le__(self, other):  # is self <= other?
        return not self > other  # this calls self.__gt__(other)

    def __lt__(self, other):  # is self < other?
        return not (self > other or self == other)

    def __ge__(self, other):  # is self >= other?
        return not self < other
```

```python
import datetime  # we will use this for date objects


class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month,
                                 self.birthdate.day):
            age -= 1

        return age


person = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12),  # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)
```

EXAMPLES

```
In[2]: class Person:
   ...:     def __init__(self, name, surname):
   ...:         self.name = name
   ...:         self.surname = surname
   ...:
   ...:     def fullname(self):
   ...:         return "%s %s" % (self.name, self.surname)
   ...:
   ...: jane = Person("Jane", "Smith")
   ...:
In[3]: print(dir(jane))
['__doc__', '__init__', '__module__', 'fullname', 'name', 'surname']

In[4]:
```

- Use function **dir** for inspecting objects: output list of the attributes and methods

```
In[3]: print(person.name)
Jane
In[4]: print(person.email)
jane.doe@example.com
In[5]: print(person.age())
25
```

Exercise 1

1. Explain what the following variables refer to, and their scope:
    1. `Person`
    2. `person`
    3. `surname`
    4. `self`
    5. `age` (the function name)
    6. `age` (the variable used inside the function)
    7. `self.email`
    8. `person.email`

Answer to exercise 1

1. 1. `Person` is a class defined in the global scope. It is a global variable.
   2. `person` is an instance of the `Person` class. It is also a global variable.
   3. `surname` is a parameter passed into the `__init__` method – it is a local variable in the scope if the `__init__` method.
   4. `self` is a parameter passed into each instance method of the class – it will be replaced by the instance object when the method is called on the object with the `.` operator. It is a new local variable inside the scope of each of the methods – it just always has the same value, and by convention it is always given the same name to reflect this.
   5. `age` is a method of the `Person` class. It is a local variable in the scope of the class.
   6. `age` (the variable used inside the function) is a local variable inside the scope of the `age` method.
   7. `self.email` isn't really a separate variable. It's an example of how we can refer to attributes and methods of an object using a variable which refers to the object, the `.` operator and the name of the attribute or method. We use the `self` variable to refer to an object inside one of the object's own methods – wherever the variable `self` is defined, we can use `self.email`, `self.age()`, etc..
   8. `person.email` is another example of the same thing. In the global scope, our person instance is referred to by the variable name `person`. Wherever `person` is defined, we can use `person.email`, `person.age()`, etc..

```python
import datetime  # we will use this for date objects


class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month,
                                 self.birthdate.day):
            age -= 1

        return age

```

Exercise 2

1. Rewrite the `Person` class so that a person's age is calculated for the first time when a new person instance is created, and recalculated (when it is requested) if the day has changed since the last time that it was calculated.

Answer to exercise 2

1. Here is an example program:

```
import datetime

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

        # This isn't strictly necessary, but it clearly introduces these attributes
        self._age = None
        self._age_last_recalculated = None

        self._recalculate_age()

    def _recalculate_age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        self._age = age
        self._age_last_recalculated = today

    def age(self):
        if (datetime.date.today() > self._age_last_recalculated):
            self._recalculate_age()

        return self._age
```

Exercise 3

1. Explain the differences between the attributes `name`, `surname` and `profession`, and what values they can have in different instances of this class:

```
class Smith:
    surname = "Smith"
    profession = "smith"

    def __init__(self, name, profession=None):
        self.name = name
        if profession is not None:
            self.profession = profession
```

```
class Smith:
    surname = "Smith"
    profession = "smith"

    def __init__(self, name, profession=None):
        self.name = name
        if profession is not None:
            self.profession = profession
```

Answer to exercise 3

1. `name` is always an instance attribute which is set in the constructor, and each class instance can have a different name value. `surname` is always a class attribute, and cannot be overridden in the constructor – every instance will have a surname value of `Smith`. `profession` is a class attribute, but it can optionally be overridden by an instance attribute in the constructor. Each instance will have a profession value of `smith` unless the optional `surname` parameter is passed into the constructor with a different value.

Exercise 4

1. Create a class called `Numbers`, which has a single class attribute called `MULTIPLIER`, and a constructor which takes the parameters `x` and `y` (these should all be numbers).
    1. Write a method called `add` which returns the sum of the attributes `x` and `y`.
    2. Write a class method called `multiply`, which takes a single number parameter `a` and returns the product of `a` and `MULTIPLIER`.
    3. Write a static method called `subtract`, which takes two number parameters, `b` and `c`, and returns `b - c`.
    4. Write a method called `value` which returns a tuple containing the values of `x` and `y`. Make this method into a property, and write a setter and a deleter for manipulating the values of `x` and `y`.

Answer to exercise 4

1. Here is an example program:

```
class Numbers:
    MULTIPLIER = 3.5

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        return self.x + self.y

    @classmethod
    def multiply(cls, a):
        return cls.MULTIPLIER * a

    @staticmethod
    def subtract(b, c):
        return b - c

    @property
    def value(self):
        return (self.x, self.y)

    @value.setter
    def value(self, xy_tuple):
        self.x, self.y = xy_tuple

    @value.deleter
    def value(self):
        del self.x
        del self.y
```

Create a class called `Numbers`, which has a single class attribute called `MULTIPLIER`, and a constructor which takes the parameters `x` and `y` (these should all be numbers).

1. Write a method called `add` which returns the sum of the attributes `x` and `y`.
2. Write a class method called `multiply`, which takes a single number parameter `a` and returns the product of `a` and `MULTIPLIER`.
3. Write a static method called `subtract`, which takes two number parameters, `b` and `c`, and returns `b - c`.
4. Write a method called `value` which returns a tuple containing the values of `x` and `y`. Make this method into a property, and write a setter and a deleter for manipulating the values of `x` and `y`.

Exercise 5

1. Create an instance of the `Person` class from example 2. Use the `dir` function on the instance. Then use the `dir` function on the class.
    1. What happens if you call the `__str__` method on the instance? Verify that you get the same result if you call the `str` function with the instance as a parameter.
    2. What is the type of the instance?
    3. What is the type of the class?
    4. Write a function which prints out the names and values of all the custom attributes of any object that is passed in as a parameter.

Answer to exercise 5

1. 1. You should see something like `'<__main__.Person object at 0x7fcb233301d0>'`.
   2. `<class '__main__.Person'>` – `__main__` is Python's name for the program you are executing.
   3. `<class 'type'>` – any class has the type `type`.
   4. Here is an example program:

```
def print_object_attrs(any_object):
    for k, v in any_object.__dict__.items():
        print("%s: %s" % (k, v))
```

Exercise 6

1. Write a class for creating completely generic objects: its `__init__` function should accept any number of keyword parameters, and set them on the object as attributes with the keys as names. Write a `__str__` method for the class – the string it returns should include the name of the class and the values of all the object's custom instance attributes.

Answer to exercise 6

1. Here is an example program:

```
class AnyClass:
    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)

    def __str__(self):
        attrs = ["%s=%s" % (k, v) for (k, v) in self.__dict__.items()]
        classname = self.__class__.__name__
        return "%s: %s" % (classname, " ".join(attrs))
```

This lecture re-uses selected parts of the OPEN BOOK PROJECT

Learning with Python 3 (RLE)

http://openbookproject.net/thinkcs/python/english3e/index.html

available under GNU Free Documentation License Version 1.3)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/

This lecture re-uses selected parts of the PYTHON TEXTBOOK

Object-Oriented Programming in Python

http://python-textbok.readthedocs.io/en/1.0/Classes.html#

(released under CC BY-SA 4.0 licence Revision 8e685e710775)