COM6115: Text Processing

Programming for Text Processing:

OO Programming: Python basics

Mark Hepple

Department of Computer Science
University of Sheffield

Object Oriented Programming

- So far, we have used a *procedural programming* paradigm
  - ◇ focus is on writing *functions* or *procedures* to operate on data
- Alternative paradigm: Object Oriented Programming (OOP)
  - ◇ focus is on creating *classes* and *objects*
  - ◇ objects contain both *data and functionality*
- OOP has become the *dominant* programming paradigm
  - ◇ developed to make it easier to create and/or modify large, complex software systems
- These slides introduce *basics* of OOP in Python (*without inheritance*)
- See the '*extended presentation*' slides (on module homepage) for:
  - ◇ more on background and motivation for OOP
  - ◇ basics of using inheritance in Python

Let's talk about meaning

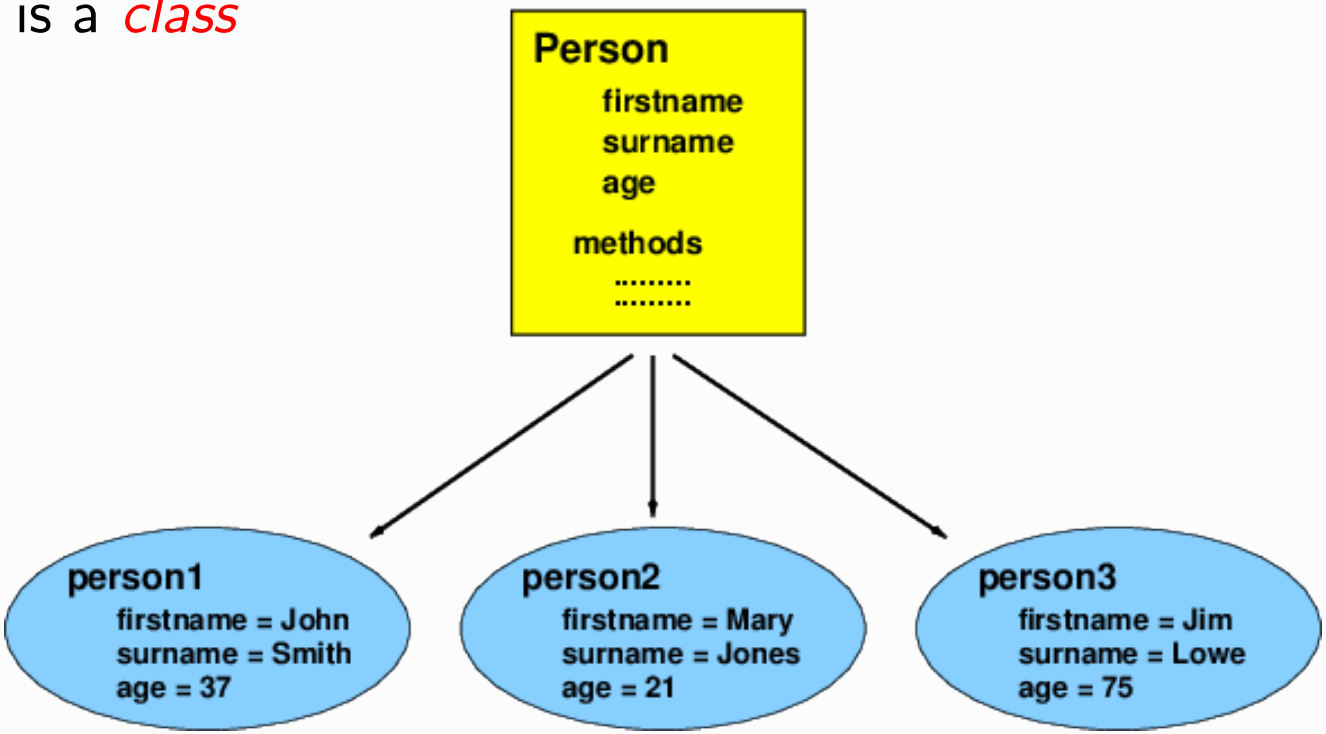
- Key notion: *CONCEPT*
  - general idea of a *class of things* with particular properties in common
    e.g. concepts: *person, bird, animal, vehicle, chair, etc.*
- A concept has *INSTANCES*
  - actual occurrences in the world
    e.g. concept *person* has *instances* such as: *Me! You! Beyoncé!*
- For a given concept, expect certain *attributes*
  - a specific *actual* person will *instantiate* these attributes
    e.g. for *person*, expect: *age, gender, height, etc.*
- Concept may also have associated *expected behaviours*
  e.g. for *person* — *walk, talk, read, hoover, give birth*
- These ideas approximate key ideas of OOP, especially:
  - concept $\approx$ *CLASS*
  - instance $\approx$ *OBJECT*

Objects and Classes — *an example*

- A Person class might:
  - ◇ have *attributes* (variables) for:
    - name, age, height, address, tel.no., job, etc
  - ◇ have *methods* (functions) to:
    - update address
    - update job status
    - work out if they are adult or child
    - work out if they pay full fare on the bus
    - *etc.*
- There might be many *objects* of the Person class
  - ◇ each representing a *different person*
    - *with different specific data*
  - ◇ but all store *similar information* and *behave similarly*

Objects and Classes — *an example* (ctd)

- Person is a *class*



  - ◇ person2, person2 & person3 are *objects*

Defining Classes in Python — *initialisation*

- Definition opens with keyword class + class name
- Class needs an *initialisation* method
  - ◇ called when an instance is created
  - ◇ has 'special' name: `__init__`
  - ◇ establishes the *attributes* (i.e. vars) belonging to objects

```
class Person:
    def __init__(self):
        self.firstname = None
        self.surname = None
        self.age = None
        self.species = 'homo sapiens'
```

  - ◇ note use of *special variable* `self` here
  - ◇ it is the instance's way of *referring to itself*

    e.g. `self.species` above means "the `species` attribute of *this instance*"

Defining Classes in Python — *creating instances*

- `Person` class with its *initialisation* method, again:

```
class Person:
    def __init__(self):
        self.firstname = None
        self.surname = None
        self.age = None
        self.species = 'homo sapiens'
```

- Can create an object (i.e. *instance*) of this class as follows:

```
>>> p1 = Person()
>>> p1.species
'homo sapiens'
```

  - ◇ here, call to `Person()` creates a new instance of the Person class
    - the `__init__` method is called automatically, to initialise the object
    - the object is assigned to `p1`

Objects (Instances) as Bundles of Data

- Last example again:

```
>>> p1 = Person()
>>> p1.species
'homo sapiens'
```

  - ◇ statement `p1.species` accesses `p1`'s `species` *attribute* directly
    - i.e. that value is accessed in the e.g. above, and printed by the interpreter

- Can think of a objects as being like "*bundles of data*"
  - ◇ each object is a different *bundle of data*, storing info about a *different instance of the class*
  - ◇ Note extra `self` arg in:

```
class Person:
    def __init__(self):
        self.firstname = None
                ...
```

    - is object's way of talking about *itself*, i.e. *the bundle that I am*
    - info stored with a `self.` attribute becomes *part of the bundle*

      — is carried around with it, and is *always available*

Initialisation with Parameters

- More generally, initialisation method can have *parameters*
  - can be used to set *initial values of attributes*

```
def __init__(self, firstname, surname, age):
    self.firstname = firstname
    self.surname = surname
    self.age = age
    self.species = 'homo sapiens'
```

  - example of creating an instance:

```
>>> p1 = Person('John', 'Smith', 37)
>>> p1.firstname
'John'
>>> p1.age
37
```

  - note `__init__` has 4 args, but 3 given when object created – *Why?*
    - first `self` is left *implicit* – stands for *this object* (i.e. *bundle of data*)
    - that object stored as `p1`, can access bundle data directly, e.g. `p1.age`

Defining Methods — *adding functionality*

- Can define (more) functions — in OOP, are known as *methods*

```
class Person:
    def __init__(self):
          ...

    def greeting_informal(self):
        print('Hi', self.firstname)

    def greeting_formal(self):
        print('Welcome, Citizen', self.surname)
```

  - ◇ as before, self appears as 1st arg of every method
    - shows that this is an object method, i.e. will be *called from an object*
  - ◇ self again refers to *this instance*, allowing access to *its own data*
    - thus, `self.firstname` above *means* value of *my* `firstname` *attribute*
    - that value, stored with this bundle of data, is accessed and used

Defining Methods (ctd)

- Example: here create two instances:

```
>>> p1 = Person('Harry', 'Potter', 12)
>>> p2 = Person('Hermione', 'Grainger', 12)
```

- Call newly defined methods from instances:

```
>>> p1.greeting_informal()
Hi Harry
>>> p1.greeting_formal()
Welcome, Citizen Potter
>>> p2.greeting_formal()
Welcome, Citizen Grainger
```

  - note that 1st `self` arg from definition again absent – *i.e. is left implicit*
  - when `p1.greeting_informal()` is called, `p1` stores an instance, and `self` aspects of definition are *about that instance*
  - thus, method calls access data (e.g. `surname`) from given instance (`p1` or `p2`), and output depends on that

Defining Methods (ctd)

- Another method ...

```
class Person:
          ...

    def greeting_age_based(self):
        if self.age < 18:
            print('Welcome, Young', self.firstname)
        elif self.age > 60:
            print('Welcome - oh Venerable', self.firstname)
        else:
            self.greeting_formal()
```

  - ◇ here see behaviour that *uses* instance data (`firstname`) and that *is conditioned on* instance data (`age`)
  - ◇ note: 'else' case *calls* another method of the instance
    - does so in form: `self.greeting_formal()`
    - uses `self`, as it is *this object's* method being used
    - but `self` is *prefixed*, not supplied as arg

Defining Methods (ctd)

- Example:

```
>>> p1 = Person('Harry', 'Potter', 12)
>>> p2 = Person('Sirius', 'Black', 38)
>>> p3 = Person('Minerva', 'McGonagall', 66)
```

  - call methods — observe behaviour is conditioned on person's age

```
>>> p1.greeting_age_based()
Welcome, Young Harry
>>> p2.greeting_age_based()
Welcome, Citizen Black
>>> p3.greeting_age_based()
Welcome - oh Venerable Minerva
```

- Have introduced *basics* of OOP in Python (*without inheritance*)
- See the '*extended presentation*' slides (on module homepage) for:
  - more on background and motivation for OOP
  - basics of using inheritance in Python