COM3110/4115/6115: Text Processing

OO Programming: Python basics
Configuring Program Behaviour
Programming Tips

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*

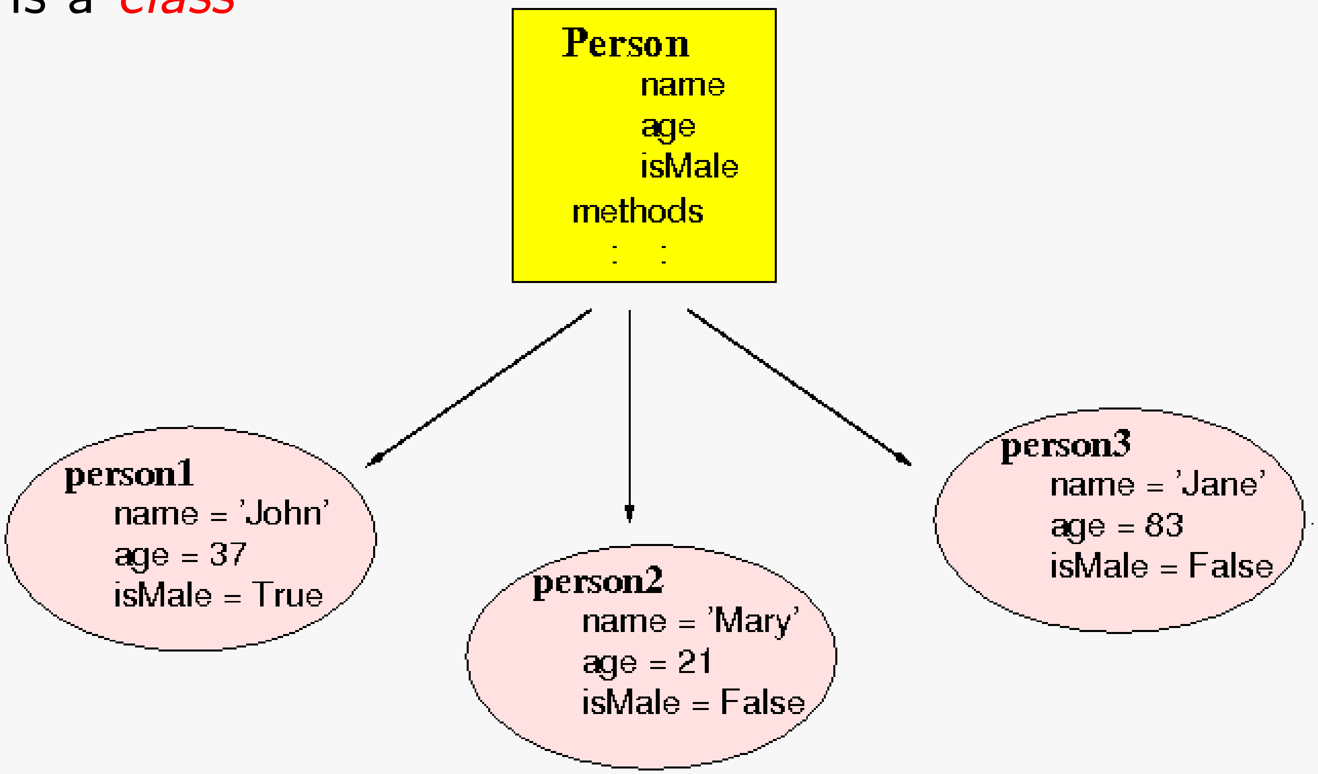
  e.g. for *person* — *walk, talk, read, hoover, give birth*
- These ideas approximate key ideas of OOP, especially:
  - concept ≈ *CLASS*
  - instance ≈ *OBJECT*

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*

- Person is a *class*



◇ person2, person2 & person3 are *objects*

Defining Classes in Python

- 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.name = None
        self.age = None
        self.species = 'homo sapiens'
        self.isMale = None
```

  - ◇ 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 (ctd)

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

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

- 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`
  - ◇ 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

Defining Classes in Python (ctd)

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

```
class Person:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.species = 'homo sapiens'
        if gender == 'm':
            self.isMale = True
        elif gender == 'f':
            self.isMale = False
        else:
            print("Gender not recognised!")
```

  - example of creating an instance:

```
>>> p1 = Person('John',44,'m')
>>> p1.name
'John'
>>> p1.isMale
True
```

Defining Classes — *adding functionality*

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

```
class Person:
    def __init__(self,name,age,gender):
        ...

    def greetingInformal(self):
        print('Hi', self.name)

    def greetingFormal(self):
        if self.isMale:
            print('Welcome, Mr', self.name)
        else:
            print('Welcome, Ms', self.name)
```

  - as before, self used to refer to *this instance*
  - allows access to this instance's *data*

Defining Classes — *adding functionality* (ctd)

- Using methods:

```
>>> p1 = Person('Harry',12,'m')
>>> p2 = Person('Hermione',12,'f')
>>> p1.greetingInformal()
Hi Harry
>>> p1.greetingFormal()
Welcome, Mr Harry
>>> p2.greetingFormal()
Welcome, Ms Hermione
>>>
```

  ◇ here, method calls both *use* the instance data (name), and show behaviour *conditioned* on that data (gender)

Defining Classes — *adding functionality* (ctd)

- Another method . . .

```
class Person:
          ...

    def greetingAgeBased(self):
        if self.age < 18:
            print('Welcome, young', self.name)
        elif self.age > 60:
            print('Welcome, venerable', self.name)
        else:
            self.greetingFormal()
```

  - ◇ note here that 'else' case *calls* another method of this instance
  - ◇ observe use of self in method call, i.e. in `self.greetingFormal()`
    - compare to the method's definition (given earlier)
    - that definition specifies a single parameter (argument) '`self`'
    - but that argument not provided in above method call
    - *instead*, it is *implicit* in the "`self.__`" prefix

Defining Classes — *adding functionality* (ctd)

- Example calls . . .

```
>>> p1 = Person('Harry',12,'m')
>>> p2 = Person('Minerva',88,'f')
>>> p3 = Person('Sirius',50,'m')
>>> p1.greetingAgeBased()
Welcome, young Harry
>>> p2.greetingAgeBased()
Welcome, venerable Minerva
>>> p3.greetingAgeBased()
Welcome, Mr Sirius
>>>
```

- 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

Configuring Program Behaviour

- Often want to *configure* the behaviour of a program, e.g. to:
  - specify files from which to take *input*
  - name of files to which to write *output/results*
  - *set* various *parameters*:

    e.g. weight/threshold values, number of results to print, etc
- For *scientific computing*, often want to run program under a wide *range* of different *settings*:
  - i.e. so alternative results can be *compared*, *plotted*, etc.
- Might configure via a GUI, *but*
  - time-consuming to develop
  - time-consuming to use, if each configuration must be entered separately
- Alternative: configure via the *command line*
  - use '*flag*' symbols (e.g. '`-s`') to *name* specific command line options

Command Line Options

- Using command line options — e.g. might have call:

  `python myCode.py -w -t 0.5 -d data1.txt -r results1.txt`

  - with options to specify the input data file (`-d`), the results file (`-r`), and a threshold value (`-t`) affecting the process
  - and a *boolean* option `-w` to direct some aspect of behaviour

    e.g. whether terms are *weighted* or not

- **Help option**: good practice to include a boolean *help* option `-h`:
  - if present, code *just prints* help message and *then quits*
  - help message says how to call program, lists options, etc

- Allows for use of *batch files*:

  i.e. text file containing commands to invoke program under a range of parameter settings

  - easy way to generate a range of experimental results

The getopt Module

- The `getopt` module helps with parsing command line options
  - allows both *short* options (`-s`) and *long* ones (`--long-option`)
  - here consider only short options
- Specify allowed options via a string, e.g. `'hi:o:I'`
  - each letter in string accepted as an option
  - letters followed by `":"` require an arg string, e.g. `-i` here
  - otherwise flag is boolean, e.g. `-h` here
- Parsing usually applied to `sys.argv[1:]`

  e.g. `opts, args = getopt.getopt(sys.argv[1:],'hi:o:I')`
  - here, `opts` is the options found — given as a *list of pairs*
    - convent to *convert* to a dictionary, e.g. with opts = dict(opts)
  - `args` is any remaining 'bare' arguments – as a list
    - flag options should *precede* bare args on command line
  - note that `sys.argv[0]` is name of your *code file* – don't pass this

The getopt Module (ctd)

- See ‘demo’ code file on using getOpts module (on module homepage)
  - run in a CMD window (or linux/mac terminal)
  - invoke python on code directly, as follows:

```
> python getOptsDemo.py -h
------------------------------------------------------------
USE: python getOptsDemo.py (options)
OPTIONS:
    -h : print this help message
    -s FILE : use stoplist file FILE (required)
    -b : use binary weighting (default is off)
------------------------------------------------------------
>  python getOptsDemo.py -s stops.txt -b file1.txt file2.txt

SUMMARY
Command line strings: ['getOptsDemo.py', '-s', 'stops.txt', '-b', 'file1.txt',
Arguments: ['file1.txt', 'file2.txt']
Options:
   Stopwords file: stops.txt
   Binary weighting: 1
>
```

Python Tips — *the Good, the Bad, and the Ugly*

- *Elegance* is important:
  - clear, readable coding helps rapid/effective code development
- Learn to use the clean constructs Python provides

  e.g. use `k in dict` rather than `dict.has_key(k)`
- Know the *default iteration* behaviour of your data structure
  - so can usually address content via a simple *for*-loop
- Understand the importance of *hash-based* data structures
  - allow *constant time* look-up / update
  - usually much more efficient than sequence-based data structures
  - *beware* of doing sequence-based look-up in hash-based structures

Python Tips — *know the default iteration behaviour*

- Simple *for*-loop provides clean, readable way to address content of an interable data structure:

```
for item in Iterable:
    do_something(item)
```

  - so, useful to know *default iteration behaviour* for *common cases*

- Iterating over X gives items Y ...
  - a string gives chars in their given (left-to-right) order
  - a list gives its elements, in their given order
  - a tuple gives its elements, in their given order
  - a set gives its elements, in no particular order
  - a dictionary gives its keys, in no particular order
  - a file-stream gives its lines of text, in file order

Python Tips — *hash-based data structures*

- In text processing, often want to handle info about *very many items*

  e.g. counts for 100K words, or *millions* of ngrams

- Hash-based data structures are very suitable for this

  i.e. Python *dictionary* and *set* data structures

- Why? — allow (roughly) *constant time* access to info for a key/item

  i.e. in a *fixed* (small) amount of time *irrespective of how many items stored*

- Using sequential data structs (e.g. list) for similar tasks is a *bad idea*

  ◇ gives (typically) *linear time* access (i.e. $\propto$ num items stored)

- Test "`item in D`" uses look-up method appropriate to `D`

  e.g. if it's a *list*, look-up is by *left-to-right sequential comparison*

  e.g. if it's a *set*, look-up uses *hash-based* method

  e.g. if it's a *dictionary*, look-up uses *hash-based* method

Python Tips — *hash-based data structures* (ctd)

- Avoid changing hash look-up to sequential one — *common error*
- If `D` is a dictionary, `D.keys()` gives a 'smart iterator' over D's keys
  - ◇ so `x in D.keys()` as efficient as `x in D` (but *less elegant*!)
- BUT all of `list(D)`, `list(D.keys())`, `sorted(D)` return a *list*
  - ◇ so (e.g.) `x in sorted(D)` is *sequential* and *v.inefficient*
- Also v.inefficient is following attempt to check for `x` in `D`:

```
for k in D.keys():
    if k == x:
        ...
```

  - ◇ recreates sequential character of look-up
  - ◇ surprisingly commonly seen!

Python Tips — *avoid piecemeal coding solutions*

- Desire to break task into manageable 'chunks' sometimes leads to *inelegant 'piecemeal' solutions*
  - ◇ avoid this, *unless the task really requires it*
- *Example*: task = count the non-stoplist words in a file
  - ◇ might be tempted to handle as follows (assume stoplist loaded):
    - read the lines of text into a list
    - iterate over list to split each line into a list of tokens
    - iterate again, to delete stop list words
    - iterate again, counting tokens (into a dictionary)

    — *this is a poor solution !!*
  - ◇ better solution — more efficient, and simpler to code:
    - read the text line by line (i.e. using a for-loop)
    - for each line read, access tokens

      e.g. using .split() string method, or using a regex+findall
    - for each token: if it's a stopword, skip it, otherwise count it