COM3110/4115/6115:

Text Processing

Programming for Text Processing:

OO Programming, Configuring Program Behaviour and Programming Tips

Mark Stevenson

Department of Computer Science
University of Sheffield

Outline

- Object Oriented Programming in Python
- Configuring behaviour with getopt
- Programming style tips

Object Oriented Programming

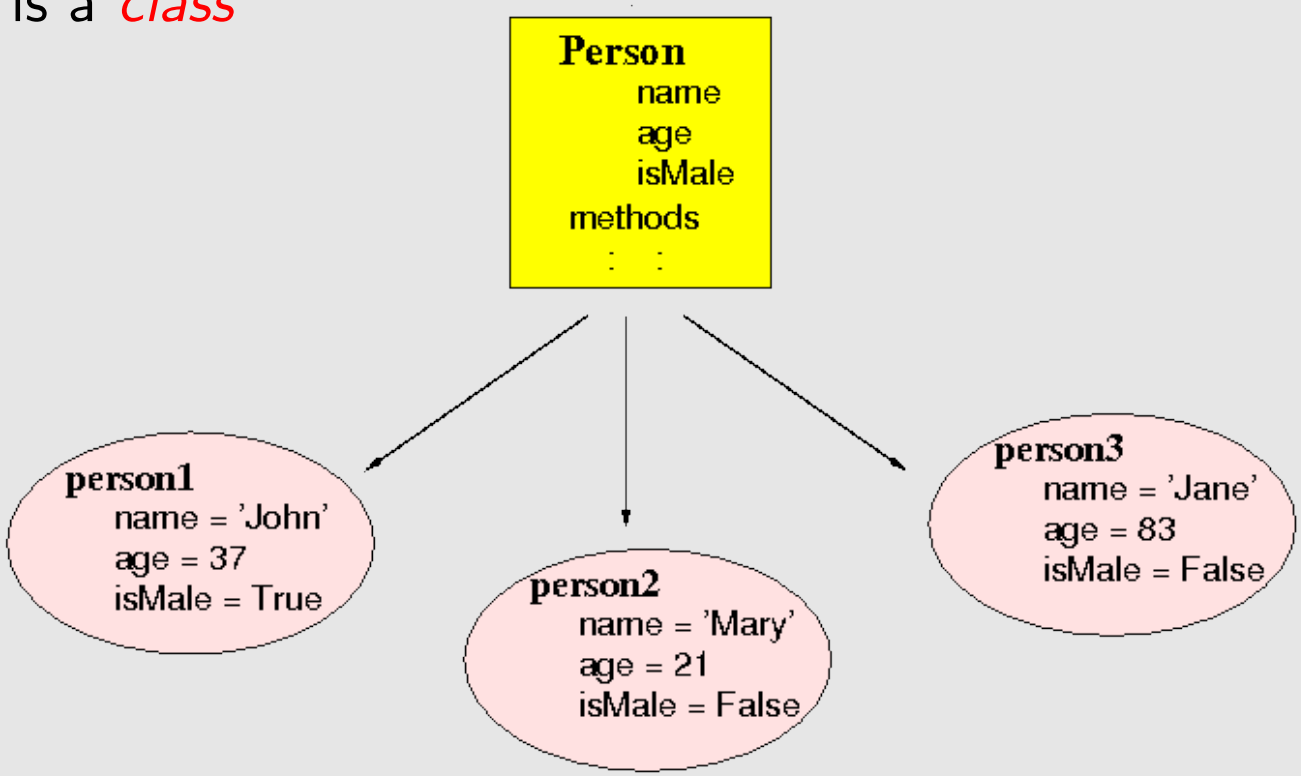
- So far, we have used a *procedural programming* paradigm
  - focus is on writing *functions* or *procedures* to operate on data
- Alternative paradigm: **Object Oriented Programming (OOP)**:
  - focus is on creating *objects*
  - objects contain both *data and functionality*
- over last 20 years, OOP has become the *dominant* programming paradigm
  - developed to make it easier to create and/or modify large, complex software systems

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*



  - ◇ person1, person2 & person3 are *objects*

Defining Classes

- 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 (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 (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!")
```

```
>>> from Person import Person
>>> 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):
        return 'Hi ' + self.name

    def greetingFormal(self):
        if self.isMale:
            return 'Welcome, Mr ' + self.name
        else:
            return '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)

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 -v -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 `-v` to direct some aspect of behavior
    - e.g. whether to print detailed (verbose) output 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

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 — list of pairs
  - `args` is any remaining 'bare' arguments – as a list
    - options should *precede* bare args on command line
  - note that `sys.argv[0]` is name of your code – don't pass this

The getopt Module - example

```
>>> 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
Arguments parsed (including command): 4
Script name: getOptsDemo.py

Program configuration
Stopwords: 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
- 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
- Avoid piecemeal coding solutions

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
- Using sequential data structs (e.g. list) for similar tasks is a *bad idea*
  - ◇ gives (typically) *linear time* access
- Avoid changing hash look-up to sequential one!
  - ◇ Example of inefficient approach:

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

  - ◇ Use x in D instead

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. for-loop)
    - for each line read, access tokens (using .split method)
    - for each token: if it's a stopword, skip it, otherwise count it

Summary

- OO programming with Python
- Configuring behaviour with `getopt`
- Programming style tips