

COM6115: Text Processing

Python Introductory Materials

Introduction

Algorithms, Control and Conditionals

Mark Hepple

Department of Computer Science
University of Sheffield

Working with the Python Interpreter

- Python is an *interpreted language*
 - ◇ can be accessed directly in *interactive mode* (a.k.a. *shell mode*)
- Can access python in shell mode in various ways:
 - ◇ using IDLE, Python's basic IDE (Interactive Development Environment)
 - ◇ directly in a terminal/CMD window, by entering "python"
 - ◇ with an advanced editor/IDE, such as *Spyder* — *recommended*
- Interactive shell at start up might look like:

```
Python 3.5.2 |Anaconda 4.1.1 (x86_64)| (default, Jul  2 2016, ..  
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on .....  
Type "help", "copyright", "credits" or "license" for more info..  
>>>
```

- In the above, ">>>" is the Python prompt
 - ◇ can enter instructions one by one at prompt
 - ◇ Python will convert and execute them

Working with the Python Interpreter (ctd)

- Simplest use: can enter arithmetic expressions
 - ◇ Python will evaluate them, and print result
 - ◇ bit like a glorified calculator

```
>>> 2 + 3
5
>>> 2.9 + 3.2
6.1
>>> 3 / 2 + 7
8.5
>>> (3.2 / 2.0) + 7
8.6
```

- the above illustrates some of Python's basic *types*:
 - ◇ integers and floats
- Arithmetic Operators:
 - plus: + minus: - divide: / multiply: *

Alternative Python Shell — the IPython Console

There's an alternative form of 'shell': [the IPython Console](#)

```
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, ...  
Type "copyright", "credits" or "license" for more information.
```

```
IPython 4.2.0 -- An enhanced Interactive Python.
```

```
?          -> Introduction and overview of IPython's features.
```

```
%quickref -> Quick reference.
```

```
help       -> Python's own help system.
```

```
object?    -> Details about 'object', use 'object??' for extra ...
```

```
%gui       -> A brief reference about the graphical user interface.
```

```
In [1]: 2 + 3
```

```
Out[1]: 5
```

```
In [2]: 3 / 2 + 7
```

```
Out[2]: 8.5
```

```
In [3]: 2.9 + 3.2
```

```
Out[3]: 6.1
```

```
In [4]:
```

Alternative Python Shell — the IPython Console (ctd)

```
In [1]: 2 + 3
```

```
Out[1]: 5
```

```
In [2]: 3 / 2 + 7
```

```
Out[2]: 8.5
```

```
In [3]: 2.9 + 3.2
```

```
Out[3]:
```

- IPython Console is widely used in ‘numeric processing’ labs
 - ◇ e.g. for maths, stats, physics, etc
- IPython Console has a few differences to standard interpreter shell
 - ◇ e.g. has some preloaded libraries
 - ◇ e.g. displays graphics ‘in-line’ in Console (rather than a separate window)
 - ◇ but mostly these differences won’t matter

Python Strings and Basic Printing

- Another key Python type is `string`
 - ◇ a `string` is just a sequence of *characters*, e.g. letters, spaces, etc
 - ◇ surrounded by *paired quote* characters
 - either single (') or double (") quotation chars

e.g. `"hello world"` and `'see this'` are both strings
- The Python `print` function will print a value to the screen
 - ◇ the value might be a string, integer or float
 - ◇ can print *multiple values*, separated by commas

```
>>> print("Hello world!")
Hello world!
>>> print("Our combined wealth is", 22.55 + 11.33, 'pounds.')
Our combined wealth is 33.88 pounds.
>>>
```

Using Variables

- Can store values using **variables**, so can reuse
 - ◇ a **variable** is like a *named bucket* which can hold values
 - can later use that bucket's name to *access* the stored value
 - ◇ can *change* the value stored in a variable — *hence the name*
- Use "=" to *assign* value to variable, e.g. statement:
 <varname> = <expression>
 - ◇ assigns result of evaluating <expression> to variable <varname>

```
>>> x = 12.5
>>> x
12.5
>>> y = 3 * x + 2
>>> y
39.5
>>> x = x + 1.1
>>> x
13.6
```

Using Variables (ctd)

- It's good practice to use *meaningful* variable names
 - ◇ helps to make your code more *readable*
 - ◇ makes it easier to spot *errors* in your code
 - ◇ an important practice for being a good programmer

e.g. *converting* distance in *miles* to distance in *kilometers*

- where each kilometer is $\frac{5}{8}$ of a mile

```
>>> miles = 28.5
>>> kilometers = (miles * 8.0) / 5.0
>>> print("Distance in miles:", miles)
Distance in miles: 28.5
>>> print("Distance in kilometers:", kilometers)
Distance in kilometers: 45.6
>>>
```


Storing Code for Reuse

- There are many **tasks** that we may want to *repeat*
 - ◇ commands entered to Python **shell** are *'lost to time ...'*
 - ◇ better to *store* the commands in a **file** for later *reuse*
- Example: use **Spyder** as an editor, to create a file with contents:

```
# Converts distance in miles to kilometers
miles = 22.7
kilometers = (miles * 8.0) / 5.0
print("Converting distance in miles to kilometers:")
print("Distance in miles:      ", miles)
print("Distance in kilometers:", kilometers)
```

- ◇ such a file is called a *code file* or *script*
- ◇ line beginning “#” is a *comment* — it is ignored by the interpreter
- ◇ with the code open in Spyder, can run it just by pressing F5
 - NB: on *first run*, should press “ctrl + F6” to configure run, so a fresh python interpreter is used

Storing Code for Reuse (ctd)

- *Example continued ...*

- ◇ For this example, when the code is run, we see the following in the interpreter window:

```
>>>  
Converting distance in miles to kilometers:  
Distance in miles:      22.7  
Distance in kilometers: 36.32  
>>>
```

- ◇ might now use editor to set different values for the miles variable, and re-run to compute a range of results

Defining Functions

- Can store commands in a code file for later reuse
- BUT often want to be able to reuse some functionality *within* the same program
 - ◇ this is achieved by *defining a function*
- Function definitions use **keyword** **def** and have following form:

```
def NAME( LIST-OF-PARAMETERS ):  
    BLOCK-OF-STATEMENTS
```

where:

- ◇ NAME is the name of the defined function
- ◇ LIST-OF-PARAMETERS allow us provide inputs to the function
 - **parameters** sometimes referred to as *arguments*
 - list may be *empty*
- ◇ BLOCK-OF-STATEMENTS is some series of commands
 - whose limit is identified by *indentation*

Defining Functions (ctd)

- For example, a function to convert distances in miles to kilometers might be:

```
def convertDistance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    print('Distance in miles:', miles)  
    print('Distance in kilometers:', kilometers)
```

where:

- ◇ the function defined has *name* `convertDistance`
- ◇ input value (no. of miles) is provided via a *parameter*
- ◇ the block of statements that make up the *body* of the definition are all *indented*
 - shows that they belong to the *body* of the definition

Defining Functions (ctd)

- To *call* this function, we:
 - ◇ invoke it by name
 - ◇ in an appropriate context (where it will be evaluated)
 - ◇ and with a series of *argument values*
 - one for each parameter
 - arguments within parentheses, immediately after function name (no space between name and open bracket)
 - NB: parentheses needed even if there are *zero* arguments/parameters

e.g. might call as:

```
>>> convertDistance(10)
Distance in miles: 10
Distance in kilometers: 16.0
>>>
```

Defining Functions (ctd)

- The above example function *prints* its results
- More commonly want a function to *return* its result
i.e. so this can be used elsewhere
- Use keyword *return* to do this, e.g.

```
def convertDistance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    return kilometers
```

- Then, can *use result value* for some further purpose
e.g. to assign to some variable, or use in a calculation

```
>>> k = convertDistance(10)  
>>> k  
16.0  
>>> 2 + convertDistance(10)  
18.0  
>>>
```

Indentation

- Note the importance of *indentation* in preceding function definitions
 - ◇ serves to indicate which command statements make up the *code block* forming the body of the function
- In the following code, for example:

```
def helloGoodbye():  
    print("Hello.")  
    print("Goodbye.")  
  
print("Au revoir.")
```

- ◇ first two print statements are indented — belong to definition
 - ◇ third print statement ('Au revoir.') is not — so, is not part of definition
- In Python, indentation is used to signal code blocks throughout
 - ◇ this is a distinctive, and somewhat controversial, feature of Python
 - ◇ helps to give Python its clean, readable syntax

Indentation (ctd)

- Most languages use *parentheses* to signal code blocks
 - e.g. braces “{..}” used in *Java* and various other languages
 - ◇ for such languages, indentation does not affect the *meaning* of code
 - although ‘good indentation’ encouraged for readability
- For Python, good indentation is *compulsory*
 - ◇ incorrectly indented code *will not run*
 - ◇ crucial, therefore, to use an *indentation aware* editor, such as Spyder
 - ◇ with such an editor, using TAB/DEL in ‘indentation area’ should move cursor in/out by ‘one level of indentation’
 - ◇ by convention, ‘one level of indentation’ corresponds to 4 spaces (but you shouldn’t have to think about that with a good editor)

Functions revisited — using `return`

- In lab, saw examples of functions that *print* their result
- More commonly want a function to *return* its result
i.e. to '*hand back*' the result, so this can be used elsewhere
- Use keyword `return` to do this, e.g.

```
def convertDistance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    return kilometers
```

- Then, can assign result to some variable, e.g.

```
>>> k = convertDistance(10)  
>>> k  
16.0  
>>>
```

Functions revisited — using `return` (ctd)

- Python functions *always* return a value
 - ◇ even if there's no explicit `return` command
 - ◇ in that case, just returns `None` (special Python 'null' value)
- Executing a `return` command, always *terminates* the function call
 - ◇ hence, makes no sense to have code such as:

```
def myfunction():  
    return 1  
    return 2
```

- second `return` command would never be reached
- Sometimes, may use `return` for specific *purpose* of terminating function execution
- **Note:** can use `return` on its own
 - i.e. without a return value being specified
 - ◇ in this case, function returns `None` by default

Functions revisited — using `return` (ctd)

- It's important to realise that `print` cannot be used in place of `return`
 - ◇ **WARNING:** a common mistake is for students not to grasp this
 - ◇ Following definition has `print` instead of `return`

```
def convertDistance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    print(kilometers)
```

- ◇ behaviour may sometimes appear similar, e.g.:

```
>>> convertDistance(10)  
16.0
```

- ◇ but value just sent to screen — can't get hold of it for further use

```
>>> k = convertDistance(22)  
35.2  
>>> k  
>>> print(k)  
None
```

Programs and Algorithms

- A computer *program* is a set of instructions that tell a computer how to carry out a task
- The instructions are written in a special *formal* language
 - ◇ a *programming language*
- In order to solve a programming problem, we need a step-by-step specification of the actions that must be taken to compute result
 - ◇ this specification is called an *algorithm*
- A *algorithm* should be:
 - ◇ *precise* and *unambiguous*
 - ◇ *correct*, i.e. finish and deliver correct result
 - ◇ *efficient*, but this depends on task
- Same *algorithm* can be implemented in different languages, or even stated in (pseudo) *English*
 - i.e. algorithm idea more general than specific piece of code

Algorithms — Examples

- Algorithms can be expressed in English-like ‘**pseudocode**’
- **Task:** making a cup of (instant) coffee

```
1. Fill kettle
2. Boil kettle
3. Put spoon of coffee in cup
4. Fill cup (nearly) with water from kettle
5. Add a dash of milk
```

- This ‘algorithm’ is just a single fixed sequence of actions
- Can handle more complex tasks by allowing:
 - ◇ **conditionals:** actions that happen only under certain conditions
 - ◇ **loops:** (groups of) actions that repeat over until result achieved

Algorithms — Examples (ctd)

- Task: supermarket shopping

1. Get a trolley
2. While there are items on shopping list
 - 2.1 Read first item on shopping list
 - 2.2 Get that item from shelf
 - 2.3 Put item in trolley
 - 2.4 Cross item off shopping list
3. Pay at checkout

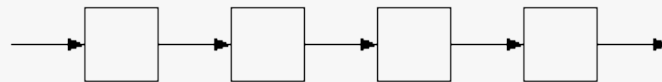
Algorithms — Examples (ctd)

- Task: supermarket shopping *on a budget!!*

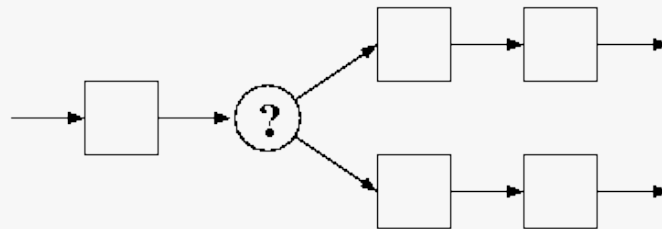
1. Get a trolley
2. While there are items on shopping list
 - 2.1 Read first item on shopping list
 - 2.2 Get that item from shelf
 - 2.3 IF item costs less than £3
 - 2.3.1 Put item in trolley
 - 2.4 ELSE
 - 2.4.1 Put item back on shelf
 - 2.5 Cross item off shopping list
3. Pay at checkout

Control Structures

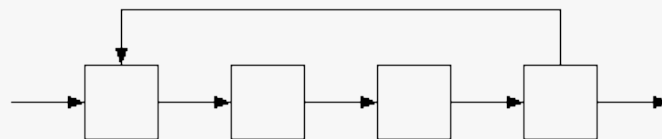
- The way that program execution moves from statement to the next is called the *flow of control* within a program
- The major control structures are **sequence**, **selection** and **repetition**:
 - ◇ **Sequence**: simply do one statement after the next



- ◇ **Selection**: flow of control is determined by a simple decision



- ◇ **Repetition**: execute a statement or block of statements more than once



Boolean type and expressions

- We have seen Python basic types such as `integer` and `float`
- A further basic type is `boolean`
 - ◇ has only *two values*: `True` and `False`
- A *boolean expression* is one that evaluates to `True` or `False`
 - ◇ the 'decision' of a *selection structure* is typically formulated as a *boolean expression*
- Simple boolean expressions commonly involve a *comparison operator*:

<code>==</code>	: <i>equal to</i>
<code>></code>	: <i>greater than</i>
<code>>=</code>	: <i>greater than or equal to</i>
<code><</code>	: <i>less than</i>
<code><=</code>	: <i>less than or equal to</i>
<code>!=</code>	: <i>not equal to</i>

Boolean type and expressions (ctd)

- For example:

```
>>> 3 == 3
True
>>> "this" == 'this'
True
>>> 3 >= 4
False
>>> 3 >= 2
True
>>> 5 != 3
True
>>> 5 != 'some string'
True
>>>
```

- *Beware*: it's easy to use “==” in place of “=”, and *vice versa*
 - ◇ a very common coding error

Boolean type and expressions (ctd)

- Can form *more complex conditions* by using the *boolean operators*:
 - ◇ they are: *and*, *or* and *not*
- Given boolean expressions E1 ,E2 then:
 - ◇ E1 and E2
 - is *True* if *both* E1 *and* E2 are *True*, and *False* otherwise
 - ◇ E1 or E2
 - is *True* if *either* E1 *or* E2 is *True*, and *False* otherwise
 - ◇ not E1
 - is *True* if E1 is *False*, and *False* otherwise
- Example: testing for teenagers!

```
>>> age = 15
>>> isaTeen = age >= 13 and age <= 19
>>> isaTeen
True
>>> age = 22
>>> isaTeen = age >= 13 and age <= 19
>>> isaTeen
False
```

Conditionals

- Selection control structures achieved by use of *if-else* constructions
 - ◇ known as *conditionals*
- Key form:

```
if CONDITION:  
    CODE-BLOCK-1  
else:  
    CODE-BLOCK-2
```

- Example:

```
if age >= 18:  
    print("Congratulations!")  
    print("You're an adult!")  
else:  
    print("Even better!")  
    print("You're an child!")
```

Conditionals (ctd)

- The *else* is *optional* — can be omitted, e.g. in:

```
if altitude < 100:  
    print("Warning!")  
    print("Time to bail out.")
```

- Can also *chain* a series of cases, using keyword *elif*, e.g.:

```
if age < 13:  
    print('child')  
elif age < 18:  
    print('teen')  
elif age < 65:  
    print('adult')  
else:  
    print('pensioner')
```

Conditionals (ctd)

- Note in preceding case that the *order of the cases* matters:
 - ◇ reordering them would give incorrect behaviour
 - ◇ consider what would happen with code having reordered cases:

```
if age < 65:  
    print('adult')  
elif age < 18:  
    print('teen')  
elif age < 13:  
    print('child')  
else:  
    print('pensioner')
```