introduction

February 8, 2015

Contents

0.1 Scientific Programming in Python

February 2015

0.2 Session 1: Basics

(based on slides by Loic Matthey, UCL Graduate School)

0.2.1 Agenda

- Brief overview of the course
- Python installation
- Basic elements of Python programming
- Python as a calculator
- Working with strings
- Working with lists

0.2.2 Introduction

Things to know about programming

- programming is not science
- programming is a skill
- programming takes a long time to master (longer than this week!)
- programming is a tool commonly used by scientists, engineers and artists

What is programming?

- The process of creating computer programs.
- Everything running on your computer is a program.
- Every program is a piece of specifically written text, called the "source code".
- Programming is simply writting text, which will get executed by your computer.

When is programming useful?

- Automating a task, when they are repetitive or error prone
- Analysing data, creating plots automatically
- Mathematics/simulation heavy tasks

What is Python?

- Created to be general-purpose, high-level programming language.
- A "fun to use" language, hence named after Monty Python's Flying Circus
- Combines "remarkable power with very clear syntax" and provides a large and comprehensive standard library
  - The "standard library" is a collection of code already written and directly available to you.
  - That's a good thing: it means people have written code for most applications you could be thinking of.
- Multi-paradigm programming language permitting several styles:
  - object-oriented
  - structured
  - functional
  - aspect-oriented
- Application domains: web development, database access, desktop GUIs, education, scientific and numeric computing, network programming, game and 3D graphics

Why learn Python?

- very clear and readable syntax
- extensive standard libraries and modules for virtually every task
- high-level, easy to use datatypes, such as flexible arrays and dictionaries
- interpreted language - no compilation is necessary
- Easy to switch to another language later

I know Matlab, why Python?

- Free.
- More versatile, you can use it for different applications.
- Input and output file manipulation better in Python.
- Speedwise, they are about the same (when using Numpy/Scipy appropriately)

Python installation

Check: http://www.cs.ucl.ac.uk/scipython/help.html

HELP!

- type `help()`, e.g. `help(sum)` will tell you how to use the Python function called `sum`
- the official Python website: `www.python.org`, and tutorial: `docs.python.org/tutorial`
- attend one of the PyCon conferences, e.g. PyCon UK

Python interpreter

- A better option when using Python is to launch `ipython`
- It is a slightly changed interpreter, containing multiple tweaks and modifications to make your life easier.

  - For example, it has colors by default,
  - a great history,
  - command completion
  - and a way to launch scripts directly from it (%run, more on scripts later)
- help/ documentation:
  - typing `?` behind a function is an alias for `help()`, e.g.: `sum?` is the same as `help(sum)`.
  - typing `??` behind a function shows the source code of the function

Ipython Notebook

- This page is a rendered (static) version of an IPython notebook.
- interactive script environment inspired by Mathematica notebooks, that is run in a webbrowser
- allows to write text, $LaTeX$ equations (something like $\sqrt{\frac{3}{x}}$), together with python code
- very good way to do quick, but organized prototyping and developping
- to start the ipython notebook interface launch it with `ipython notebook` in the terminal, this will:
  - start an IPython kernel and a local http server that hosts the session locally
  - open a webbrowser showing you the IPython notebooks in the folder you ran the command from
  - then you can organize your notebooks with the web-browser and do all development

0.2.3 Python basics

- the interpreter acts as a simple calculator, e.g.

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

```
Out[3]: 4
```

- parentheses can be used for grouping, e.g.

```
In [2]: (50 - 5*6)/4
```

```
Out[2]: 5
```

- integer division returns the floor:

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

```
Out[3]: 2
```

- if one of the operands is a float, then the remaining operands will automatically be converted into floats, e.g.

```
In [4]: 7.0 / 2
```

```
Out[4]: 3.5
```

Operations

Operator
Operation
+
plus
-
minus
*

multiplication
/
division
**
power (not ˆ as in some other languages)
%
moldulo (remainder of integer division)
==
equal to (do not mix up with =)
!=
not equal

greater than

<

smaller than

=

greater than or equal to

<=

smaller than or equale to

If you want more complex operations (e.g. `sin`, `sqrt`, `log`), you'll need to use `math` or `numpy`. More on that later.

Operators precedence

The priority between operators is the same as you'd expect mathematically. From highest to lowest precedence:
Operator
Operation
()
Terms in backets
**
Exponentiation
+x, -x
Unary plus and minus
*, /, %
Multiply, divide, modulo
+, -
Addition and subtraction
<=, <, >, >=
Comparison operators
==, !=
Equality operators
not, or, and
Logical operators
There are other operators that you will encounter later, check the Python Library Reference for the complete precedence table.

Variables

- a variable are how values are stored in a computer
- a variable can be a number, a character, a collection of numbers, etc.
- the equal sign (`=`) is used to assign a value to a variable (again, be careful to not confuse with testing for equality `==`)

```
In [61]: width = 20
         pi = 3.14
         message = 'I have a bad feeling about this'
```

A value can be assigned to several variables simultaneously:

```
In [6]: x = y = z = 20
        x

Out[6]: 20

In [7]: y

Out[7]: 20

In [8]: z

Out[8]: 20
```

Variables must be "defined" (assigned a value) before they can be used, or an error will occur:

```
In [9]: n


        ---------------------------------------------------------------------------
    NameError                                 Traceback (most recent call last)

        <ipython-input-9-fe13119fb084> in <module>()
    ----> 1 n


        NameError: name 'n' is not defined
```

Variables are manipulated and combined with operators, forming "expressions".

```
In [5]: width = 20
        height = 5
        width * height

Out[5]: 100
```

Functions

More complex operations are provided by functions. For example, the absolute value of a number:

```
In [62]: abs(-3)

Out[62]: 3
```

- `abs()` is an example of a function.

- Functions have a name (here "`abs`"), followed by parentheses. They can take some input (here a number), do something with them and create some output (here it returns the absolute value of the number).
  - Their name should usually be self-explanatory.
  - But you can type `help(abs)` or `abs?` to see what any function does exactly.
- They form the fundamental basis of any imperative programming language.
  - Just think of them as providing you with advanced functionalities for free for now.
  - We will cover more about functions as we move along the lectures
- It is very easy to implement your own functions to structure your code or to reuse functions that have been written by others (like the `abs()` function!)

Lets write a very simple function to calculate the area of a rectangle given as: $Area = width * height$

```
In [4]: def area(width, height):
            '''Calculates the area of a rectangle'''
            return width * height

        # Let's use the function!
        area(10,5)
```

```
Out[4]: 50
```

Errors

Syntax errors If you type something into the pythonmistake and python can't understand then that's a syntax error.

- Making mistakes like this is normal. It doesn't mean that you are dumb or that the computer is dumb.
- Computers are more precise than humans. When you make a mistake like this, you have to fix it.
- You will learn how to look at an error message and figure out what the mistake is. This is a valuable skill.

```
In [63]: n = 50 - 5*6)/4
```

```
      File "<ipython-input-63-af371d7d80f2>", line 1
    n = 50 - 5*6)/4
                ^
SyntaxError: invalid syntax
```

- On the first line of python's output, it says the file and line number where the mistake is.
- On the second and third line, python prints the part of the line where the mistake is and a ^ under the mistake.
- The last line tells us what type of mistake it is: invalid syntax means the code you wrote is not proper 'grammar' in the python language.

runtime error Sometimes the grammar of your code is right and python knows what it should do, but you've asked it to do something impossible. This is called a runtime error.

```
In [64]: n = 0
         1 / n
```

```
    ---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-64-158600a3143e> in <module>()
      1 n = 0
----> 2 1 / n


    ZeroDivisionError: integer division or modulo by zero
```

- In this example, we've asked python to divide one by zero, which is impossible. The error message let's us know what went wrong.
- The first line says the file and line where the impossible command was.
- The second line says the impossible command.

logic error

- Sometimes code is grammatically correct, but it doesn't do what you want. This is called a logic error.
- As long as it's not impossible to do what your code says, python will do it anyway even though it's not what you want (how would Python know?) 8 In the example below, the programmer wanted the variable `even_number` to contain an even number, and the variable `zero` to contain the number 0.

```
In [65]: even_number = 57
         sample_size = '20'
```

Summary of errors

- **Syntax error**: your code is not 'grammatically' correct
- **Runtime error**: you've asked python to do something impossible
- **Logic errors**: your code doesn't actually mean what you think it should.

Errors in code are called bugs, and getting rid of them is called debugging. We will teach you how to debug throughout the course.

Variables

Variables in Python are "typed". Every variable has a given type, which you can check by writing `type(varname)`.

```
In [67]: a = 10
         b = 20.0
         c = "It's just a flesh wound!"
         type(a)

Out[67]: int

In [68]: type(b)

Out[68]: float

In [69]: type(c)

Out[69]: str
```

Directly combining variables of different types usually creates a `TypeError`. Python can automatically convert some of them, e.g. `int` to `float`. (Python is dynamically strongly typed)

```
In [70]: a / b

Out[70]: 0.5

In [71]: a + c
```

```
        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-71-ca57d551b7f3> in <module>()
    ----> 1 a + c


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Modules in Python

- Module is a collection of definitions.
- In order to use modules in your script or interpreter, you need to import them, e.g.:

```
In [10]: import math
```

Functions of imported modules can be accessed with a `.`, such as:

```
In [33]: math.sqrt(3)

Out[33]: 1.7320508075688772
```

You can import specific definitions from a module, e.g.:

```
In [11]: from math import sqrt
         sqrt(5)

Out[11]: 2.23606797749979
```

You can also import all the definitions from a specific module, e.g.:

```
In [12]: from math import *
```

This way is, however, **disencouraged** because it clutters your namespace (an exception might be build-in modules, like math). If two modules that you import have a function that has the same name, you cannot be sure which function you are calling, nor can readers of your code! As an example:

```
In [15]: help(sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

If we import the `numpy` package which comes with a `sqrt()` function as well, we overwrite the old definition:

```
In [16]: from numpy import *

In [31]: sqrt?
```

If you do not want to write out module names to access their functions, you can import an entire module by importing it with an alias. A better way of importing the numpy module is:

```
In [27]: import numpy as np
         from math import *

In [29]: sqrt?

In [30]: np.sqrt?
```

Exercises Write formulae to convert:

1. temperature from Celcius to Farenheit
2. weight from pounds into kilograms.

```
In [36]: celsius = 37
         fahrenheit = celsius  * (9.0/5) + 32
         fahrenheit

Out[36]: 98.60000000000001

In [41]: lb = 77.161
         kg = lb / 2.2046
         kg

Out[41]: 35.0
```

But what if we wanted to calculate fahreheit or kilograms with different values? We would have to change the values by hand every time. Instead we can write functions for the conversion:

```
In [7]: def celsius_to_fahrenheit(celsius):
            '''Converts celsius to fahrenheit'''
            return celsius * (9.0/5) + 32

        celsius_to_fahrenheit(38)

Out[7]: 100.4

In [9]: def pound_to_kilogram(lb):
            '''Converts pounds to kilogram'''
            return lb / 2.2046
        pound_to_kilogram(10)

Out[9]: 4.535970244035199
```

Strings

- a string is a sequence of characters
- strings must be enclosed in quotes (to distinguish them from variable names)
- certain characters must be escaped by backlashes, e.g. "\ "

```
In [42]: "doesn\'t"

Out[42]: "doesn't"
```

certain characters are “invisible”, e.g. use "\n" to indicate the end of a line

Strings and print

The `print` statement produces a more readable output for strings, e.g.

```
In [48]: hello = "This is a rather long string containing\n several lines of text just as you would do i
         print hello
```

```
This is a rather long string containing
 several lines of text just as you would do in C.
         Whitespace at beginning of the line is  significant.
```

- spaces are also characters:

```
In [50]: a = "dorota"
         len(a)
```

```
Out[50]: 6
```

```
In [52]: b = "dorota "
         len(b)
```

```
Out[52]: 7
```

- two (or more) strings can be merged into one by using ”+“:

```
In [53]: b + a
```

```
Out[53]: 'dorota dorota'
```

- elements of a strings can be subscripted (indexed)
- the first character of a string has subscript (index) 0

```
In [54]: word = 'Help'
         word[1]
```

```
Out[54]: 'e'
```

- substrings can be specified with the slice notation: two indices separated by a colon
- the first index is inclusive, the terminating second index is exclusive

```
In [55]: word[0:2]
```

```
Out[55]: 'He'
```

- indices may be negative numbers, to start counting from the right:

```
In [56]: word[-2]
```

```
Out[56]: 'l'
```

Omitting a slice index means “start from beginning” or “end with (inlcuding) the last index”:

```
In [57]: word[-2:]
```

```
Out[57]: 'lp'
```

```
In [58]: word[:-2]
```

```
Out[58]: 'He'
```

Python strings cannot be changed - assigning to an indexed position in the string results in an error:

```
In [59]: word[0] = 'x'
```

```
        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-59-d89b9f9f38d7> in <module>()
    ----> 1 word[0] = 'x'


        TypeError: 'str' object does not support item assignment
```

Lists

What are Lists? Compound data type used to group together other values

```
In [60]: list1 = ["first", "second", "third"]
         list2 = [1, 2, 3]
         list3 = ["pi", 3.1415926, "answer", 42]
         emptyList = []
```

- Values don't have to be of the same type
- Lists can be modified after creation (they are *mutable*)
- Elements can be changed, added, and deleted
- Lists are versatile and are used extensively in typical Python code