

# **Python for Beginners**

**By**  
**Edward Smith and Katerina Michalickova**

**18th September 2017**

# Introduction

## Plan for Today

Why learn Python

10:15 to 12:30 Python introduction with Katerina

12:30 to 13:30 Lunch break

13:30 to 14:15 Types, floats, integers and functions

14:15 to 14:45 Strings and reading files

14:45 to 15:15 List and iterators

15:15 to 15:45 Dictionaries, Numpy arrays and classes

15:55 to 16:00 Summary and wrap up

<http://tinyurl.com/ichpcclass>

## Pros and Cons of Python (vs e.g. MATLAB)

### Pros

- Free and open-source
- Not just for scientific computing
- Great libraries (One of Google's languages)
- Clear, clever and well designed syntax
- Remote access (ssh)
- Great online documentation (stackoverflow!)

### Cons

- No debugging GUI so less user friendly
- Syntax is different with some odd concepts
- No type checking can cause problems
- Not as many scientific toolboxes as MATLAB, inbuilt help not as good
- Slow compared to low level languages

# Libraries

- Key numerical libraries: NumPy, matplotlib, SciPy, Pandas
- Graphical User Interfaces (GUI) e.g. Tkinter, wxpython, pyGTK, pyQT
- Multi-threading and parallel e.g. Subprocess, MPI
- Image and video manipulation e.g. pyCV, PIL
- Machine learning e.g. Scikit-learn, Pybrain
- Build system e.g. scons, make using os/system
- Differential equations solvers e.g. FEniCS, Firedrake
- Databasing and file storage e.g. h5py, pysqlite
- Web and networking e.g. HTTPLib2, twisted, django, flask
- Web scraping – e.g. scrapy, beautiful soup
- Any many others, e.g. PyGame, maps, audio, cryptography, etc, etc
- Wrappers/Glue for accelerated code e.g. HOOMD, PyFR (CUDA)
- It is also possible to roll your own

## Computing at Imperial

- Aeronautical Engineering – **MATLAB** in "Computing" and "Numerical Analysis"
- Bio-Engineering – **MATLAB** in "Modelling for Biology"
- Chemical Engineering – Only **MATLAB** taught
- Chemistry – **Python** taught
- Civil Engineering – **MATLAB** in "Computational Methods I and II" (some object oriented in second year)
- Computing/Electrical Engineering – low level
- Materials – **MATLAB** in "Mathematics and Computing"
- Maths – **Python** in 2nd term (**MATLAB** in 1st)
- Mechanical Engineering – Only **MATLAB** taught
- Physics – Start 1st year "Computing Labs" with **Python**
- Biology – Some **Python** teaching
- Medicine – No programming?

## My Background

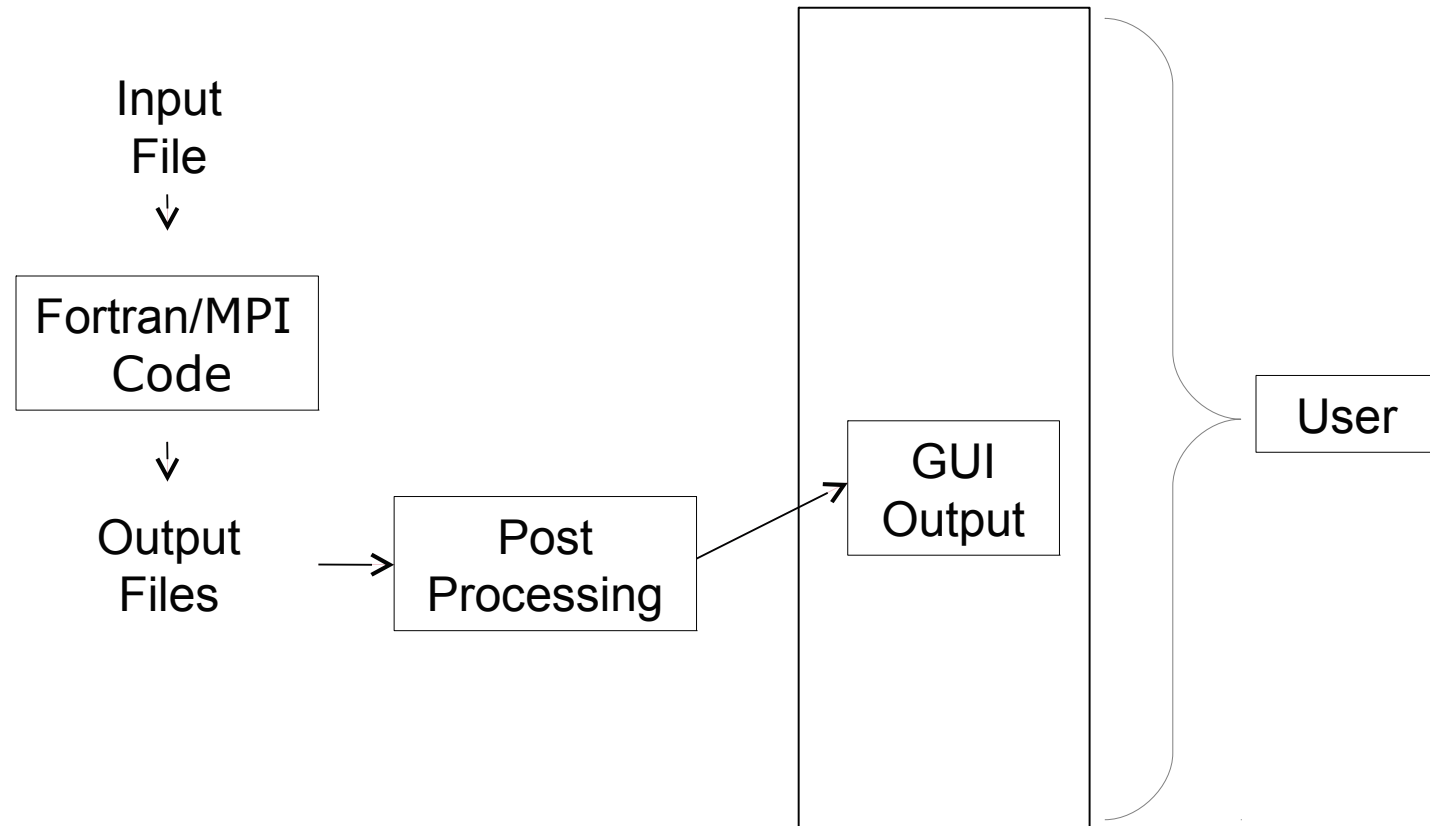
- Currently a full time software developer/researcher
  - Civil Engineering (Prev Mech & Chem Eng at IC)
  - About 8 years of programming experience
  - Software Sustainability Fellow ([www.software.ac.uk](http://www.software.ac.uk))
  - Answer Python questions on Stackoverflow
- Why this course?
  - I learnt **MATLAB** as undergrad in Mech Eng (also C++ and assembly language but still mainly used excel)
  - Masters project: Lattice Boltzmann solver in **MATLAB**. PhD: Fortran/MPI Molecular Dynamics, **MATLAB** post processing
  - Collaborator used **Python** and too much effort to maintain both
  - My main incentive for the switch to **Python** is the long term potential and the ability to write more sustainable code, but it took me a year to kick the **MATLAB** habit
  - I wish I had learnt **Python** sooner!

## How I use Python in my Work

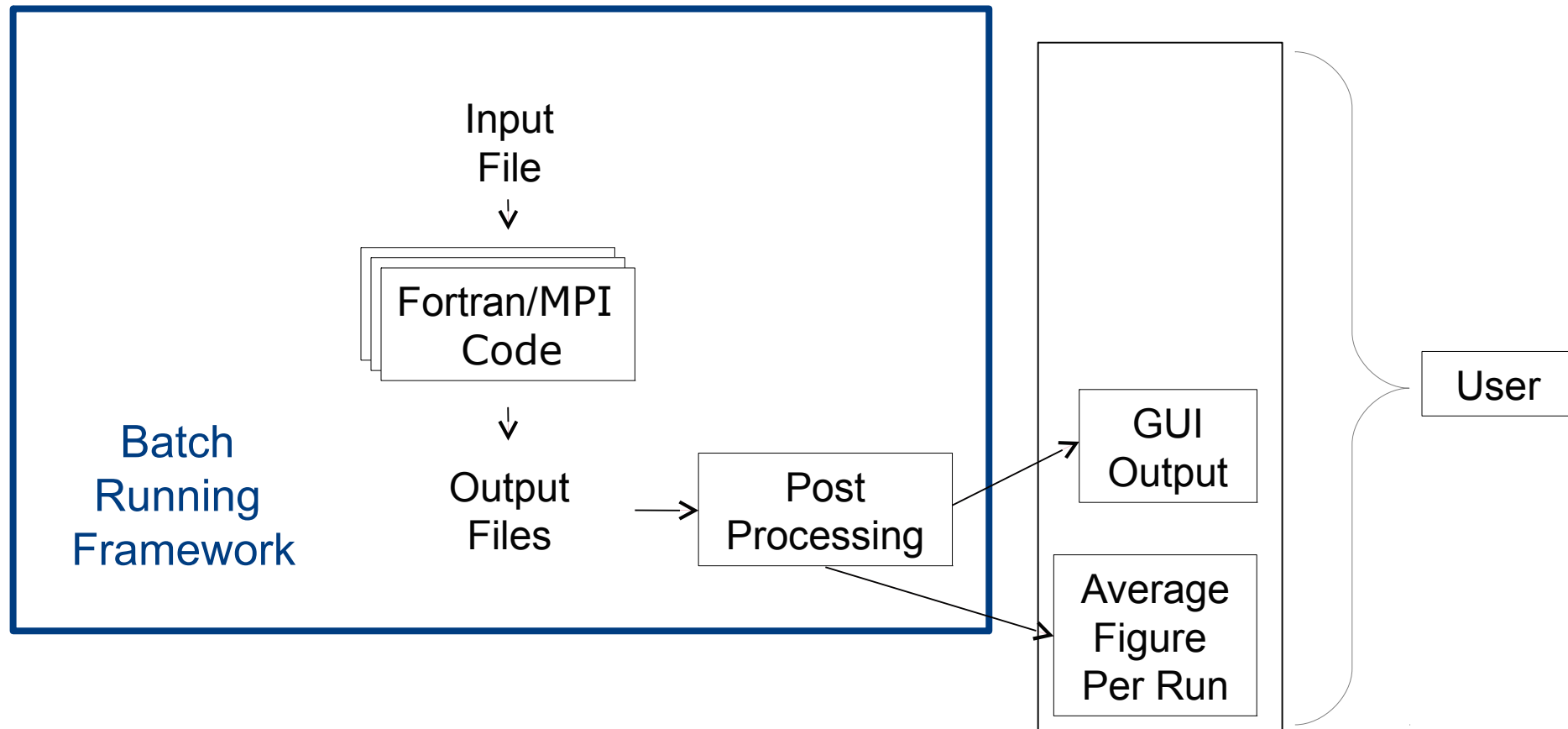
- Post processing framework
  - Low level data readers for a range of different data formats
  - Higher level field provide standard data manipulation to combine, average and prepare data to be plotted
- Visualiser Graphical User Interface
  - Read all possible field objects in a folder
  - Based on wxpython and inspired by MATLAB sliceomatic
- Batch running framework for compiled code
  - Simple syntax for systematic changes to input files
  - Specify resources for multiple jobs on desktop, CX1 or CX2
  - Copies everything needed for repeatability including source code, input files and initial state files



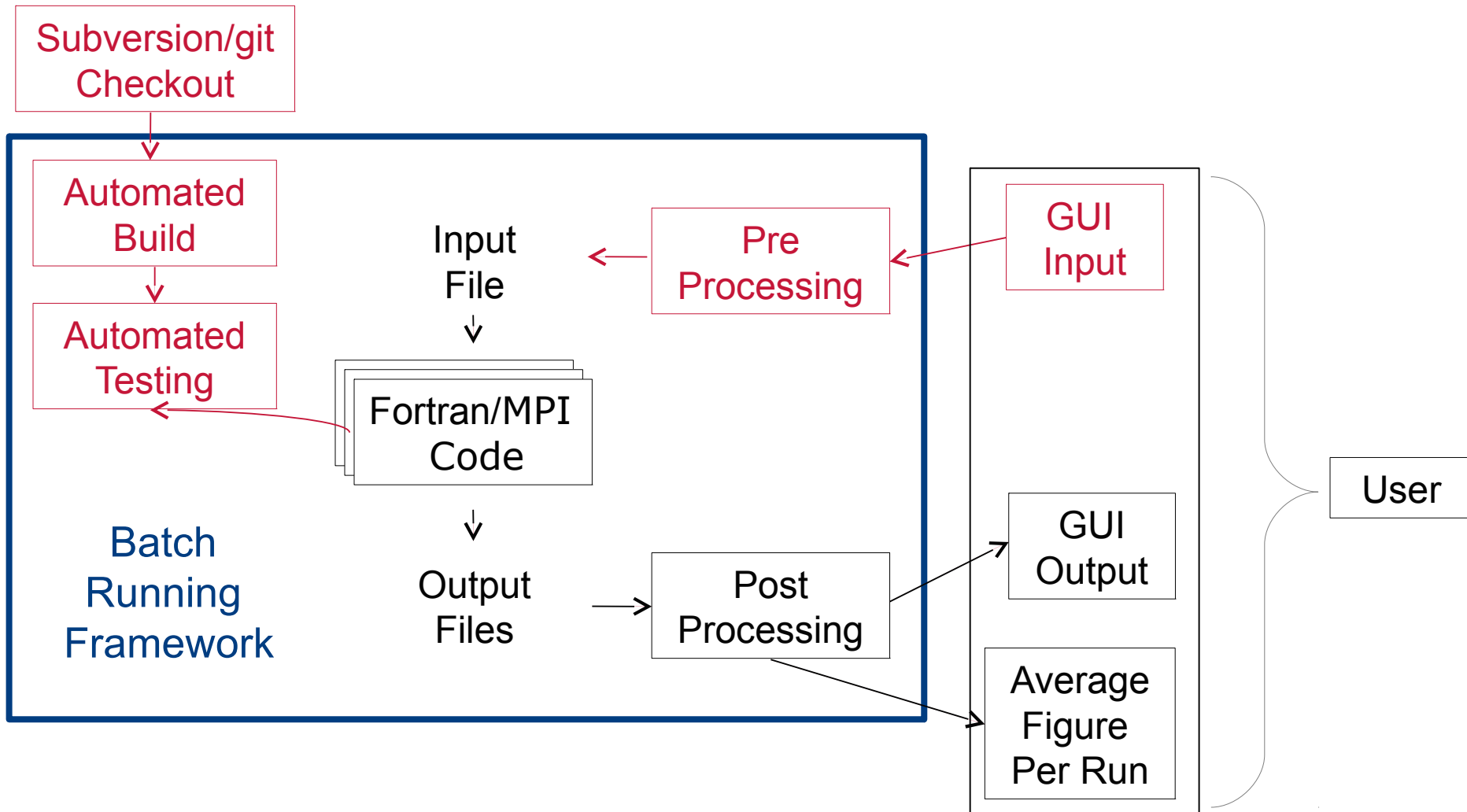
## How I use Python in my Work



# How I use Python in my Work



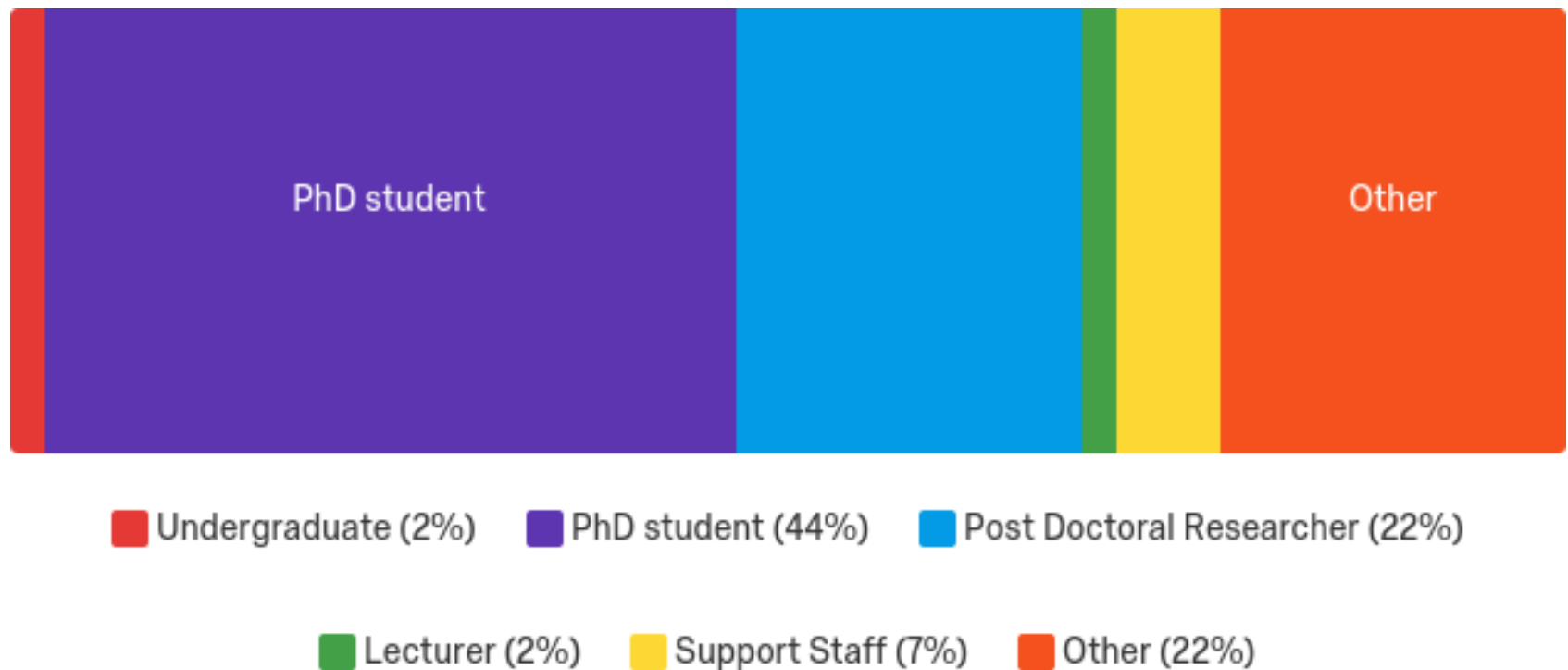
# How I use Python in my Work



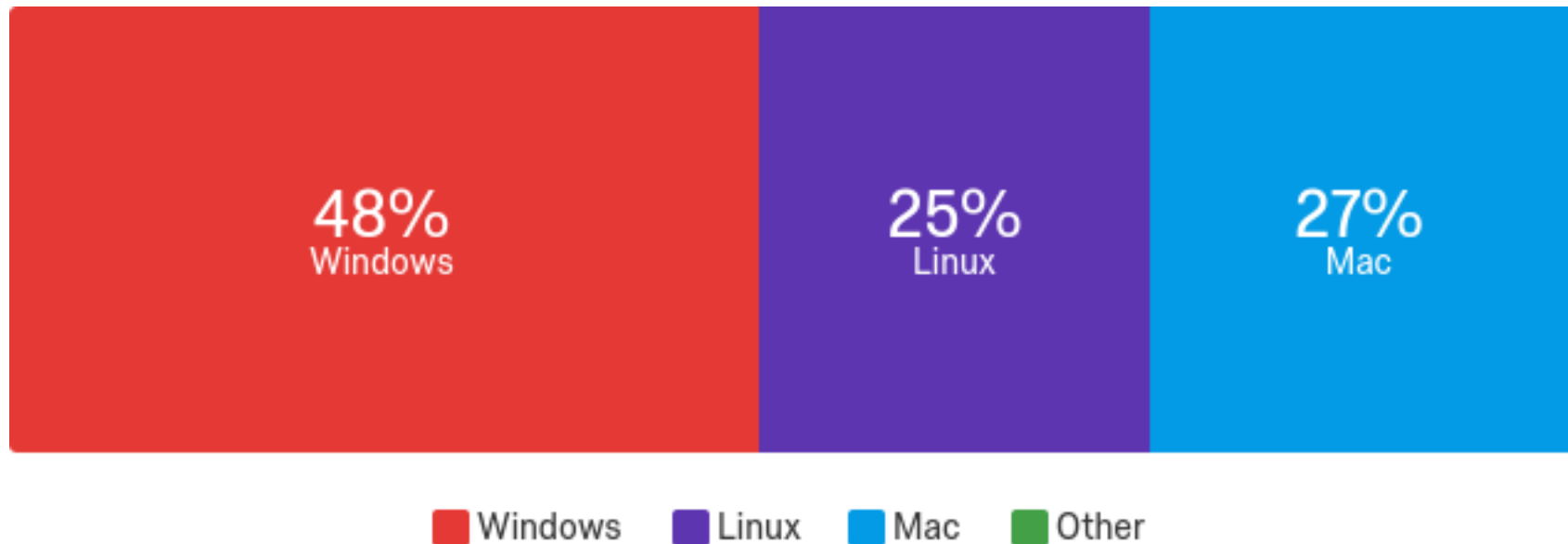
```
graph TD
    A[Subversion/git Checkout] --> B[Automated Build]
    B --> C[Automated Testing]
    C --> D[Batch Running Framework]
    D --> E[Input File]
    E --> F[Fortran/MPI Code]
    F --> G[Output Files]
    G --> H[Post Processing]
    H --> I[Results To Website]
    H --> J[GUI Output]
    H --> K[Average Figure Per Run]
    L[Pre Processing] --> E
    M[Machine Learning?] --> L
    N[Web Scrapping] --> L
    O[GUI Input] --> L
    P[GUI Run Info] --> M
    Q[User] --- O
    Q --- P
    Q --- J
    Q --- K
```

The flowchart illustrates the Batch Running Framework. It begins with 'Subversion/git Checkout' leading to 'Automated Build', which then leads to 'Automated Testing'. The 'Batch Running Framework' is a central component that receives input from 'Automated Testing' and 'Pre Processing'. It processes 'Input File' into 'Fortran/MPI Code', which then produces 'Output Files'. These output files are then processed by 'Post Processing', which generates 'Results To Website', 'GUI Output', and 'Average Figure Per Run'. The 'Pre Processing' step is influenced by 'Web Scrapping', 'GUI Input', and 'Machine Learning?'. The 'Machine Learning?' step is influenced by 'GUI Run Info'. The 'User' interacts with the 'GUI Input', 'GUI Run Info', 'GUI Output', and 'Average Figure Per Run'.

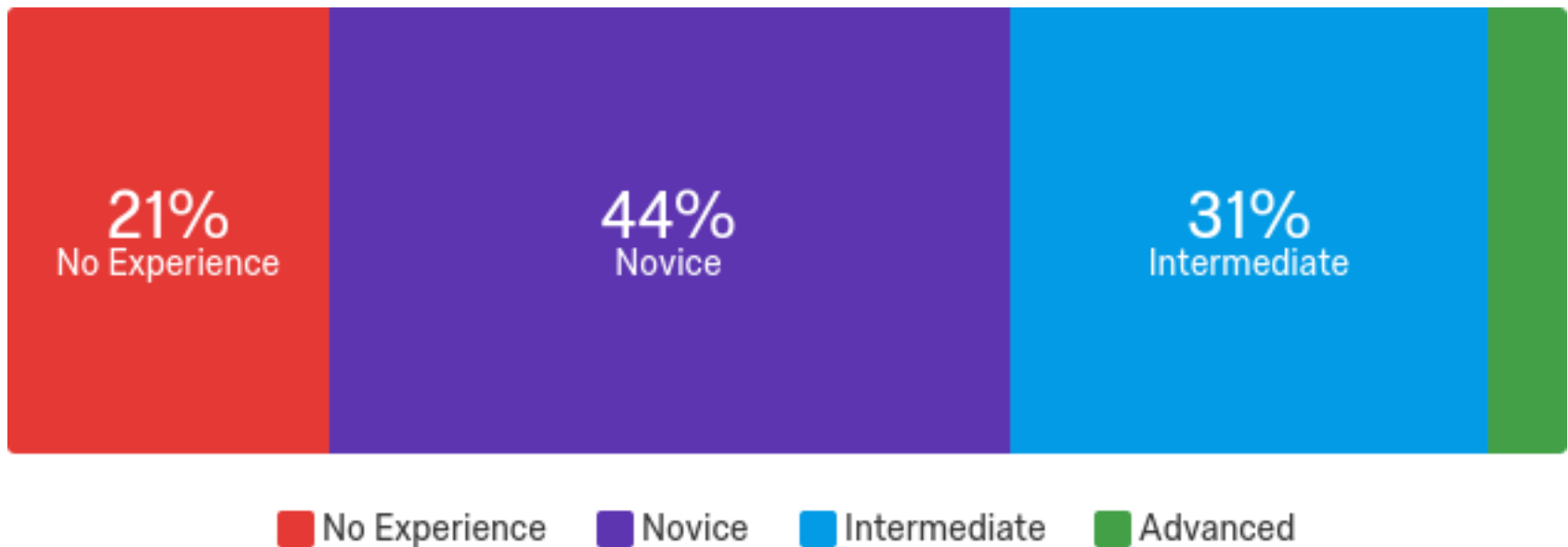
## Who you are



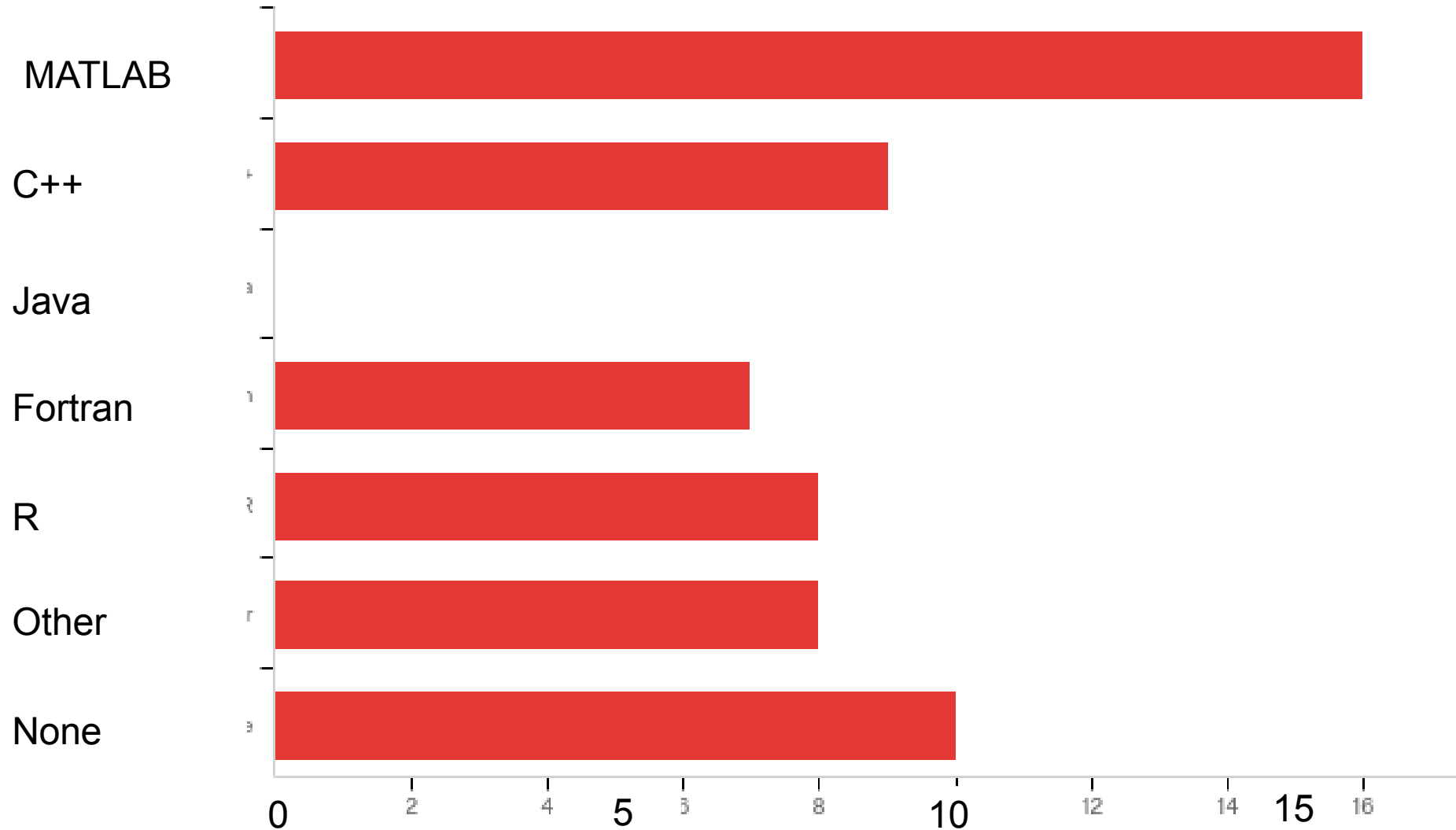
## Your Operating System



## Your Programming Experience

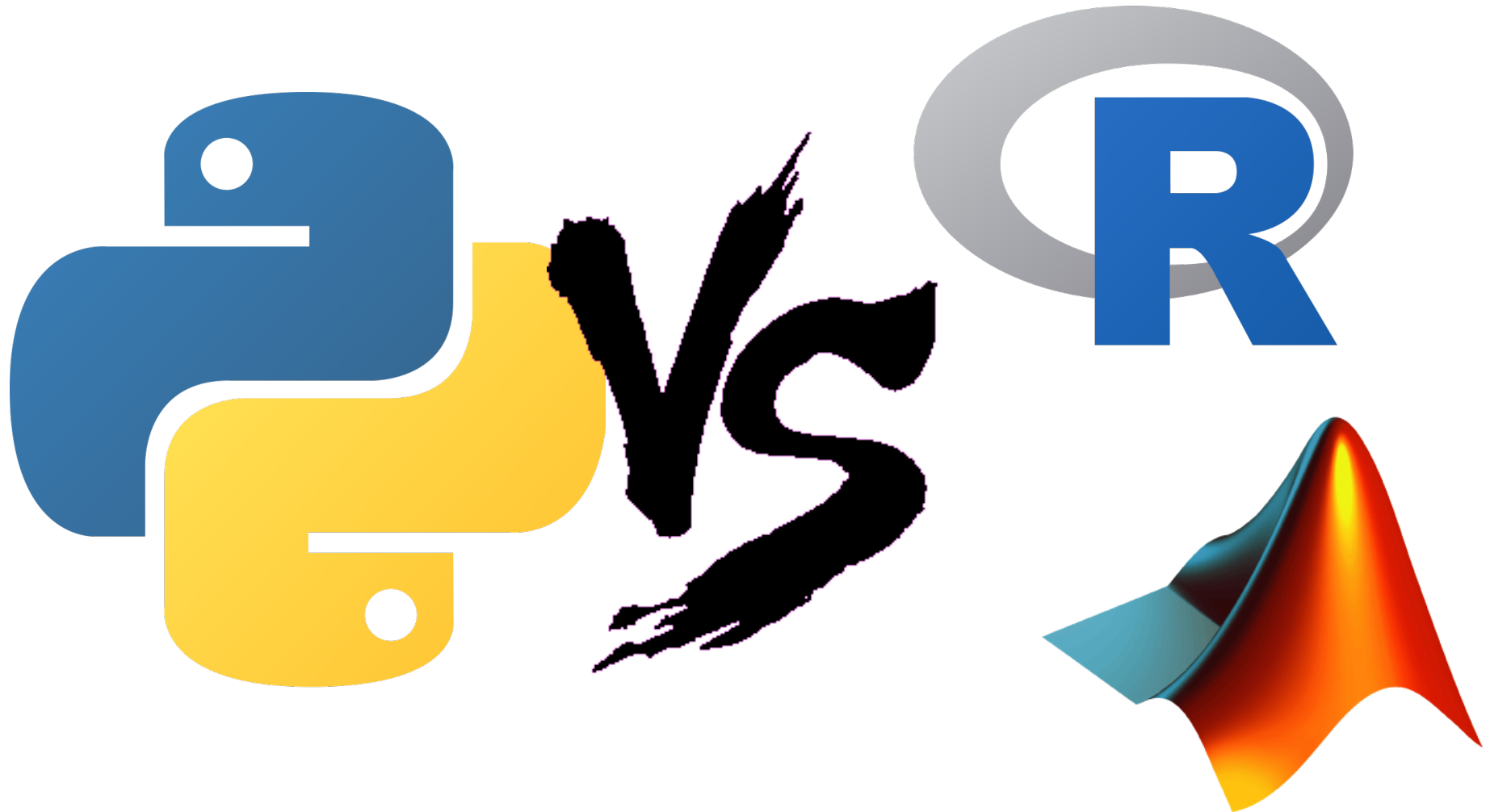


## Your Language Background





## Python VS MATLAB (and R?)



## An Example vs MATLAB

%MATLAB

clear all

close all

x = linspace(0,2\*pi,100);

y = sin(x);

z = cos(x);

plot(x,y,'-r');

hold all

plot(x,z,'-b')

#python

from numpy import \*

from matplotlib.pyplot import \*

x = linspace(0,2\*pi,100)

y = sin(x)

z = cos(x)

plot(x,y,'-r')

plot(x,z,'-b')

show()

## Some of your aims for the course

- I would like to know how to extract data from output files on a programme .. plot graphs and perform more calculations ... interface python ... automate the simulations I am performing.
- Currently i use a python program to run some custom made lab machinery, I would like to learn the basics of python and programming so that I can start learning to write/modify my own programs.
- Importing data of various kinds and carrying out basic analysis/visualisation Writing scripts for automating some tasks i.e. converting between file formats etc
- Programming principles in Python. I am especially interested in applications in data analysis.
- 1. How other people use Python for data science. 2. What other existing online tools can help with research. 3. Best practice.
- Better understanding on the key uses, capabilities, and applications of python.

[illegible]

# **Python for Beginners**

**By  
Edward Smith**

**18th September 2017**

## Plan for Today

---

- 13:30 to 14:15 Types, floats, integers and functions
- 14:15 to 14:45 Strings and reading files
- 14:45 to 15:15 List and iterators
- 15:15 to 15:45 Dictionaries, Numpy arrays and classes
- 15:55 to 16:00 Summary and wrap up

## Overview

- An introduction to the unique features of programming in Python
- Structured around types in Python
  - Duck-typing and functional interface
  - String manipulations in Python and reading files
  - Lists and iterators (loops)
  - Dictionaries and Classes
- Minimal discussion of plotting and data analysis
  - Covered in detail tomorrow

## **My aims for the course**

- A focus on the strange or unique features of Python as well as common sources of mistakes or confusion
- Help with the initial frustration of learning a new language
- Prevent subtle or undetected errors in later code
- Make sure the course is still useful to the wide range of background experiences



## **The Zen of Python, by Tim Peters** (import this)

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

....

# Running Python

- Python is the language – various ways of parsing it
  - Open python from start menu
  - Run a script with `python script.py` from terminal (Linux/Mac) or `python.exe script.py` from Windows shell or cmd prompt
  - `ipython` – slightly more interactive version of python
  - Python Jupyter notebooks – Web browser hosted Python session either locally on your computer or website (e.g. Azure)
  - The `pythoneverywhere` website – creates a Python terminal
- Common features
  - Python code will work the same on all platforms
  - May be issues with plots backends and packages

## Summary of Today

- Show how to use the command prompt to quickly learn Python
- Introduce a range of data types (Note everything is an object)

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]             # List, note square brackets tuple if ()
d = {"red":4, "blue":5}  # Dictionary
x = np.array([1,2,3])    # Numpy array
```

- Show how to use them in other constructs including conditionals (**if** statements) iterators (**for** loops) and functions (**def** name)
- Introduce external libraries numpy and matplotlib for scientific computing

# **Floats, Integers and Functions**

## Key Concepts - Types

- Use the python command prompt as a calculator

```
3.141592653589*3.0      Out: 9.424777961
```

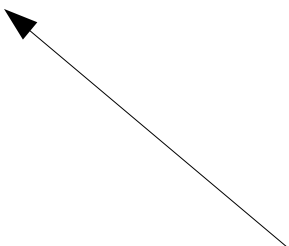
## Key Concepts - Types

- Use the python command prompt as a calculator

```
3.141592653589*3.0      Out: 9.424777961
```

- Define variables as one of several types

```
a = 3.141592653589      # Float  
i = 3                    # Integer
```



Syntax here means: “Define  
a to be 3.141592653589”  
and “define i to be 3”

## Key Concepts - Types

- Use the python command prompt as a calculator

```
3.141592653589*3.0      Out: 9.424777961
```

- Define variables as one of several types

```
a = 3.141592653589      # Float  
i = 3                   # Integer
```

- We can then perform the same calculations using variables

```
a * i      Out: 9.42477796076938      # But float*integer
```

## Key Concepts - Types

- Use the python command prompt as a calculator

```
3.141592653589*3.0      Out: 9.424777961
```

- Define variables as one of several types

```
a = 3.141592653589      # Float  
i = 3                   # Integer
```

- We can then perform the same calculations using variables

```
a * i      Out: 9.42477796076938      # But float*integer  
2 / 3      Out: 0                     # WATCH OUT FOR int/int  
2.0/3.0    Out: 0.6666666666666666    # Use floats for division  
2/float(3) Out: 0.6666666666666666    # Explicit conversion  
histogram_entry = int(value/binsize) # Integer rounds up/down
```



## Python 2 vs Python 3

- The behaviour is not consistent between Python 2 and 3

2 / 3      Out: 0      # Python 2.7

2 / 3      Out: 0.6666666666666666      # Python 3.x

- What is Python 3?
  - Released in 2008 as Guido van Rossum ("creator" of the Python language) wanted to clean up Python 2.x
  - In practice some minor syntactic changes (print is a function), remove sources of errors and better handling of unicode,
  - Historically major packages (e.g. NumPy, matplotlib) not available on 3, default 2 on systems and lots of packages ported back to 2
  - Python 3 is the future of the language so best version to use
  - 2to3 tool provides automatic conversion

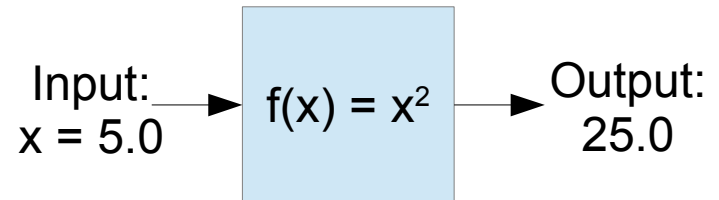
## Key Concepts - Functions

- Check type with

`type(a)`      Out: float

`type(i)`      Out: int

- `type()`, `float()` and `int()` are all examples of functions, i.e.
  - take some input,
  - perform some operation
  - return an output



## Key Concepts - Functions

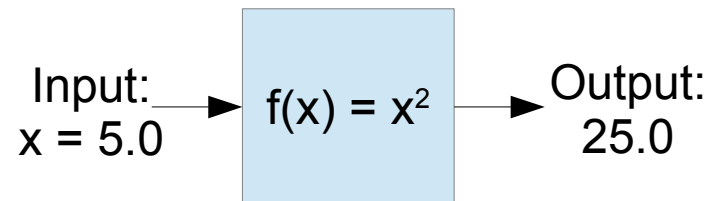
- Check type with

```
type(a)    Out: float
```

```
type(i)    Out: int
```

- `type()`, `float()` and `int()` are all examples of functions, i.e.
  - take some input,
  - perform some operation
  - return an output

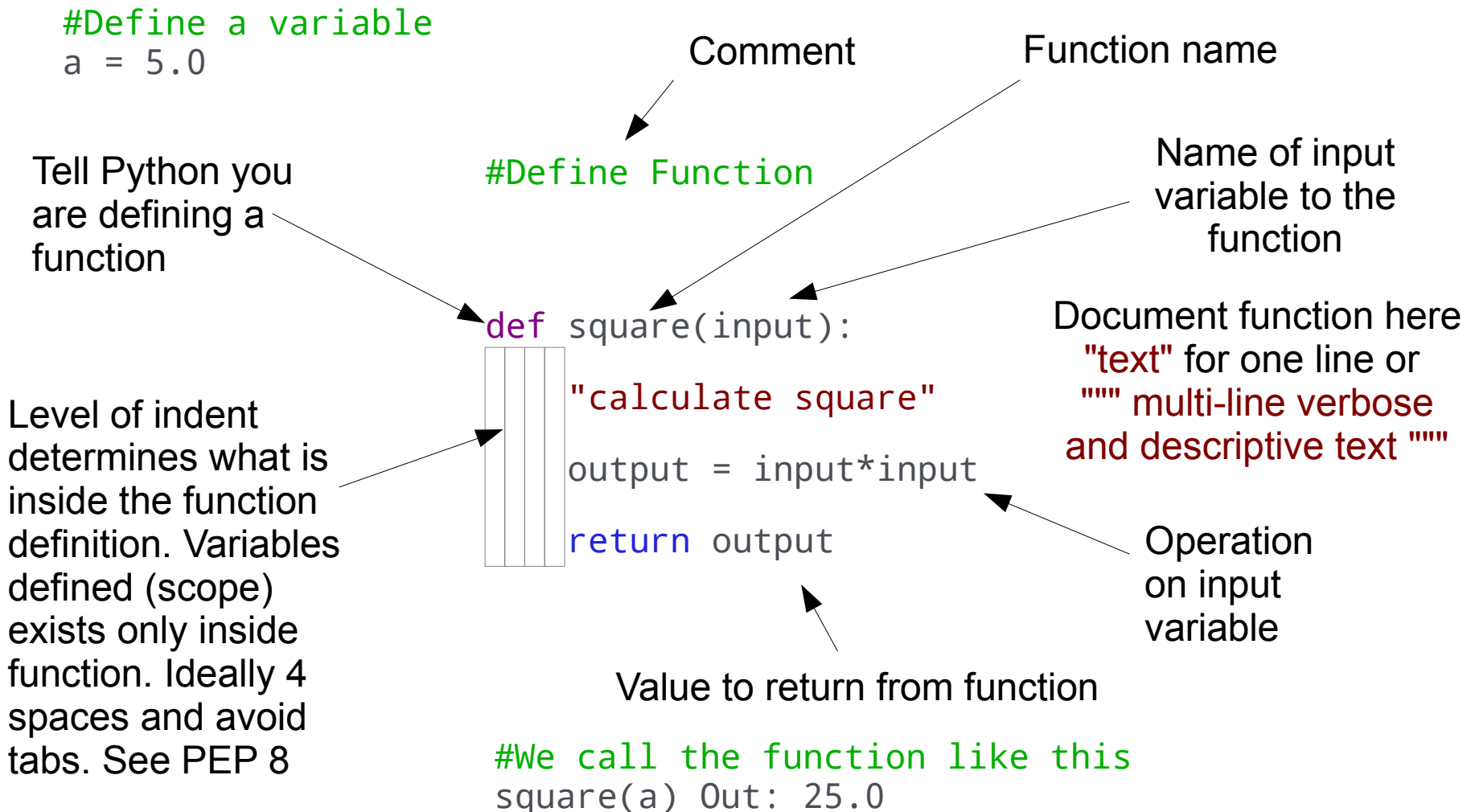
```
def square(input):  
    """Function to calculate the  
       square of a number"""  
  
    output = input*input  
    return output
```



Note: indent  
whitespace  
instead of end

```
#Now we can use this  
square(5.0) Out: 25.0
```

# Key Concepts – Function Syntax



## Key Concepts - Functions

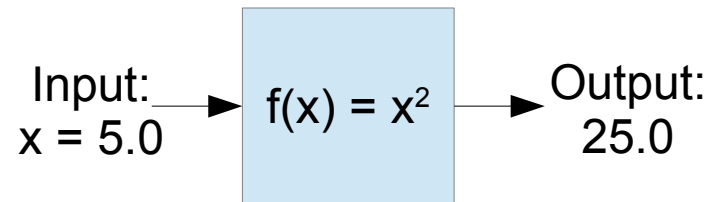
- Note that the input and output type are not specified

```
def square(input):  
    "calculate square"  
    output = input*input  
    return output
```

#Now we can use this

```
square(5.0) Out: 25.0
```

```
square(5)    Out: 25
```



## Key Concepts - Functions

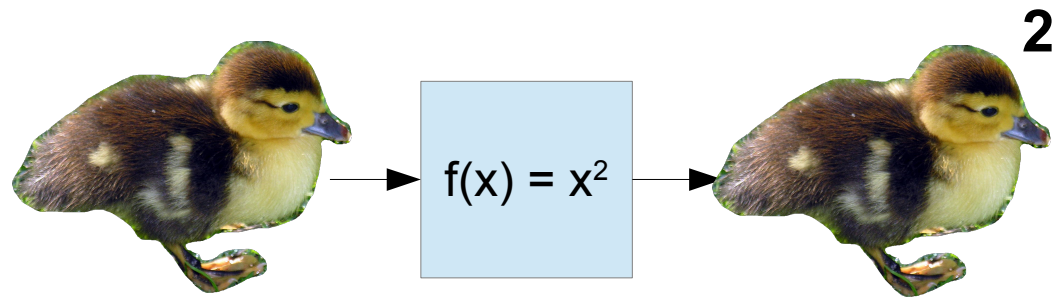
- Note that the input and output type are not specified

```
def square(input):  
    "calculate square"  
    output = input*input  
    return output
```

#Now we can use this

```
square(5.0) Out: 25.0
```

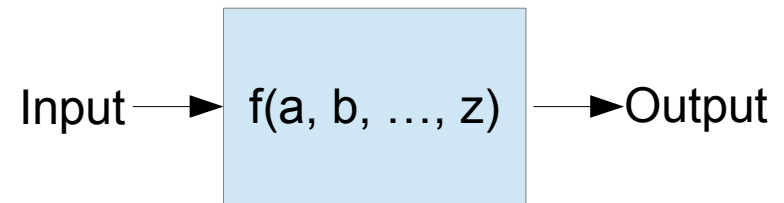
```
square(5)    Out: 25
```



- Python allows "duck typing":
  - "If it looks like a duck and quacks like a duck, it's a duck"
  - Both useful and a possible source of error
  - TypeError**: unsupported operand type(s)

# Examples of Functions

- take some inputs
- perform some operation
- return outputs



```
def divide(a, b):  
    output = a/b  
    return output
```

```
def do_nothing(a, b):  
    a+b
```

```
def get_27():  
    return 27  
  
#Call using  
get_27()
```

```
def redundant(a, b):  
    return b
```

```
def line(m, x, c=3):  
    y = m*x + c  
    return y
```

Optional  
variable.  
Given a value  
if not  
specified

```
def quadratic(a, b, c):  
    "Solve:  $y = ax^2 + bx + c$ "  
    D= b**2 + 4*a*c  
    sol1 = (-b + D**0.5)/(2*a)  
    sol2 = (-b - D**0.5)/(2*a)  
    return sol1, sol2
```

## Key Concepts - Functions

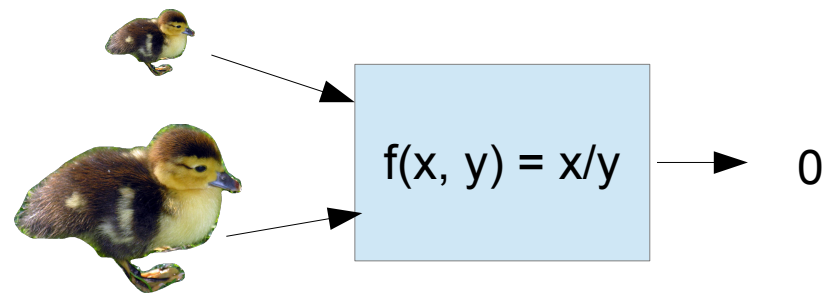
- Note that the input and output type are not specified

#Function to divide one number by another

```
def divide(a, b):  
    output = a/b  
    return output
```

#Which gives us

```
divide(2,5) Out: 0
```





## Key Concepts - Functions

- Note that the input and output type are not specified

#Function to divide one number by another

```
def divide(a, b):  
    output = a/b  
    return output
```

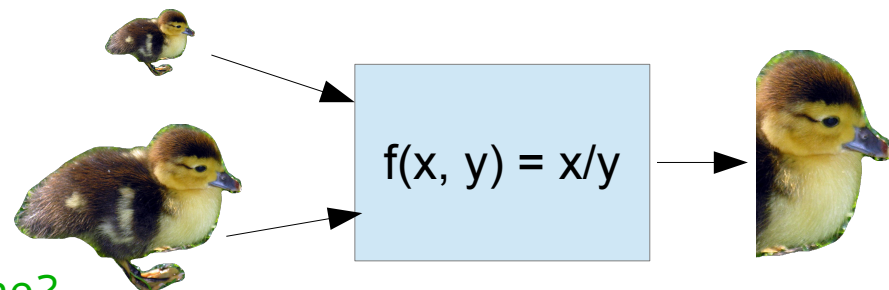
#Which gives us

```
divide(2,5) Out: 0
```

#Maybe more sensible to define?

```
def divide(a, b):  
    output = float(a)/float(b)  
    return output
```

```
divide(2,5) Out: 0.4
```



You can look at function  
information with:  
help(square) in python  
In ipython, also square? Or to  
see the code: square??

# Logical Tests

- We can write statements to test logic

#For example

```
a = 4.5
```

```
i = 5
```

```
print(type(a) is int)      # Return False
```

```
print(a < i)                # Return True
```

#These can be used in branching conditional (if) statement

```
if (a < i):
```

```
    print("a is less than i")
```

```
else:
```

```
    print("i is less than a")
```

# Conditionals

- Allow logical tests

#Example of an if statement

```
if a > b:  
    print(a)  
else:  
    print(a, b)
```

```
if type(a) is int:  
    a = a + b  
else:
```

```
    print("Error - a is type ", type(a))
```

Indent  
determine  
scope  
4 spaces  
here

Logical test to  
determine which  
branch of the  
code is run

```
if a < b:  
    out = a  
elif a == b:  
    c = a * b  
    out = c  
else:  
    out = b
```

## A Better Divide Function

- Note that the input and output type are not specified

#Add a check

```
def divide(a, b):  
    """  
    Divide a by b, a and b should be floats  
    """  
    if ((type(a) is int) and  
        (type(b) is int)):  
        raise TypeError  
    else:  
        return a/b
```

## A Better Divide Function

- Note that the input and output type are not specified

#Add a check

```
def divide(a, b):
```

```
    """
```

```
    Divide a by b, a and b should be floats
```

```
    """
```

```
    if ((type(a) is int) and
```

```
        (type(b) is int)):
```

```
        raise TypeError
```

```
    else:
```

```
        return a/b
```

- Python error handling – Better to ask forgiveness than seek permission

```
try:
```

```
    c = divide(a, b)
```

```
    print(c)
```

```
except TypeError:
```

```
    print("Cannot divide a=", a, " by b=", b)
```

## Key Concepts - Functions

- Lambda functions in Python

```
square = lambda x : x*x
```

- Resulting function is identical to

```
def square(x):  
    return x*x
```

- Functions are first class objects, we can use them as such

```
def fn(x):  
    return x+1  
  
def add_one_to_fn_output(x, some_fn):  
    return some_fn(x) + 1  
  
print(add_one_to_fn_output(x, fn))
```

## Scope in Python

- Variables defined in a function are "local" to that function

```
def fn(x):  
    xin = x + 1  
    return xin  
  
print(xin)          #Causes an error as xin local to fn
```

- But, functions in Python have access to the global scope

```
a = 3.14159  
  
def fn(x):  
    xin = x + a  
    return xin  
  
print(fn(5.))        #will print 5+3.14159
```

# Scope in Python

- Local definitions take precedence

```
a = 3.14159
```

```
def fn(x):  
    a = 2.  
    xin = x + a  
    return xin  
#will print 5.+2.  
print(fn(5.))
```

- LEGB Local, enclosing, global, builtin

```
a = 3.14159  
def outer(x):  
    a = 3.  
    def inner(x):  
        xin = x + a  
        return xin  
    return inner(x)  
#will print 5.+3.  
print(outer(5.))
```



## Part 1 Summary

- Two numerical types, floats and Integers

```
a = 2.5251
```

```
i = 5
```

You can look at function information with:

`help(type)` in python

In ipython, also `type?` Or to see the code: `type??`

- Functions allow set operations

```
def divide(a, b):
```

```
    output = a/b
```

```
    return output
```

Some Functions

`type(in)` – get type of in

`int(in)`, `float(in)` – Convert in to int, float

`help(in)` – Get help on in

- Conditional statement

```
if a > b:
```

```
    print(a)
```

```
elif a < b
```

```
    print(b)
```

```
else:
```

```
    print(a, b)
```

Design to prevent potential errors caused by Python's duck typing and lack of type checking

## Tutors

- Chris Knight
  - Isaac Sugden
  - Mark Woodbridge
  - Katerina Michalickova
  - Edward Smith
- 
- Ask the person next to you – there is a wide range of programming experience in this room and things are only obvious if you've done them before!

## Hands on session 1 – Questions

- Introduction
  - 1) Define  $a = 3.6$ ,  $b = 2$  and add them storing the result in  $c$ . What are the types of  $a$ ,  $b$  and  $c$ ?
  - 2) In a script, use an if statement to check if variable  $a$  is an integer
  - 3) Write a function to add two numbers and always return a float
  - 4) Define a function to raise a floating point number to an integer power  $N$  (Note power operator in Python to raise  $a$  to power  $N$  is  $a**N$ ). What changes are needed for non-integer powers?
- More advanced
  - 1) Write a function which combines both 2) and 4) above to get the hypotenuse of a triangle from two side lengths  $h^2 = o^2 + a^2$
  - 2) What does the function here do =====> `def add_fn(a, b, fn):`
  - 3) Write a recursive factorial function `return fn(a) + fn(b)`

# Strings and Reading Files

## Key Concepts - Types

Define variables as one of several types

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
```

# Strings

- String manipulations

```
s = "some string"
```

```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                                Out: e
```

```
s[0:4]                             Out: some
```

# Strings

- String manipulations

```
s = "some string"
```

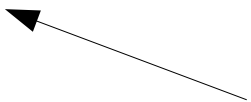
```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                                Out: e
```

```
s[0:4]                              Out: some
```

```
s.title()    Out: 'Some String'
```



Note object oriented use of a function here. Instead of `title(s)` we have `s.title()`. The object `s` is automatically passed to the `title` function. A function in this form is called a method (c.f. c++ member function)

# Strings

- String manipulations

```
s = "some string"
```

```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                Out: e
```

```
s[0:4]              Out: some
```

```
s.title()           Out: 'Some String'
```

```
s.capitalize()      Out: "Some string"
```

```
s.find("x")          Out: -1    #Not found
```

```
s.find("o")          Out: 1
```

```
t = s.replace("some", "a")    Out: t="a string"
```

- In ipython, use tab to check what functions (methods) are available



## Tab Complete

- How do we know what methods are available?

```
s = "some string"
```

```
s.title()
```

```
Out: 'Some String'
```

- In ipython, use tab to check what functions (methods) are available
- Also works on Pythoneverywhere or jupyter-notebooks
- If you are using python on cx1 then you can turn on this hidden feature as follows

```
import readline
```

```
import rlcompleter
```

```
readline.parse_and_bind("tab: complete")
```

# Strings

- String manipulations

```
s = "some string"
```

```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                Out: e
```

```
s[0:4]              Out: some
```

```
s.title()           Out: 'Some String'
```

```
s.capitalize()      Out: "Some string"
```

```
s.find("x")          Out: -1    #Not found
```

```
s.find("o")          Out: 1
```

```
t = s.replace("some", "a")    Out: t="a string"
```

- In ipython, use tab to check what functions (methods) are available

# Strings

- String manipulations

```
s = "some string"
```

```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                Out: e
```

```
s[0:4]              Out: some
```

```
s.title()           Out: 'Some String'
```

```
s.capitalize()      Out: "Some string"
```

```
s.find("x")          Out: -1    #Not found
```

```
s.find("o")          Out: 1
```

```
t = s.replace("some", "a")    Out: t="a string"
```

- In ipython, use tab to check what functions (methods) are available

# Strings

- String manipulations

```
s = "some string"
```

```
t = s + " with more"    Out: "some string with more"
```

```
s*3    Out: "some stringsome stringsome string"
```

```
s[3]                Out: e
```

```
s[0:4]              Out: some
```

```
s.title()           Out: 'Some String'
```

```
s.capitalize()      Out: "Some string"
```

```
s.find("x")          Out: -1    #Not found
```

```
s.find("o")          Out: 1
```

```
t = s.replace("some", "a")    Out: t="a string"
```

- In ipython, use tab to check what functions (methods) are available

## Files in Python

String manipulations are most useful for files

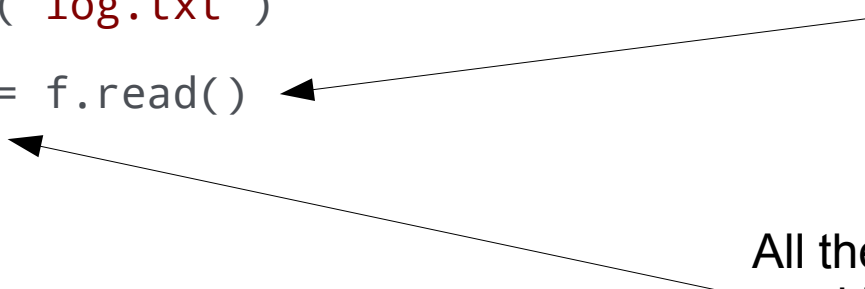
#To open a file, we use the open

# with a string for the filename

```
f = open('log.txt')
```

```
filestr = f.read()
```

Note object oriented use of a function (method). Instead of `read(f)` we have `f.read()`. The object `f` is the thing that reads the file.



All the contents of the file are read in as a string.

## Using Strings for Files

- String manipulations are most useful for files

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
f = open(fdir + './log.txt')
```

```
filestr = f.read()
```

## Using Strings for Files

- String manipulations are most useful for files


#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
f = open(fdir + './log.txt')
```

```
filestr = f.read()
```

Note object oriented use of a function (method). Instead of `read(f)` we have `f.read()`. The object `f` is the thing that reads the file.



## Best Practice for files

- String manipulations are most useful for files

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
f = open(fdir + './log.txt')
```

```
filestr = f.read()
```

```
f.close()          #Close to prevent memory leaks
```



## Best Practice for files

- Use with statement to ensure file is closed

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
with open(fdir + './log.txt') as f:
```

```
    filestr = f.read()
```

#File is automatically closed on leaving 'with' scope

## Using Strings for Files

- String manipulations are most useful for files

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
with open(fdir + './log.txt') as f:
```

```
    filestr = f.read()
```

All the contents of the file are read in as a string. This can be manipulated. E.g. if  
filestr = "contents of the file with  
some keyword=4 hidden inside"

```
"keyword" in filestr      #True if found, otherwise False
```

```
filestr.find("keyword")  #Index of "keyword"
```

## Using Strings for Files

- String manipulations are most useful for files

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
f = open(fdir + './log.txt')
```

```
filestr = f.read()
```

All the contents of the file are read in as a string. This can be manipulated. E.g. if  
`filestr = "contents of the file with some keyword=4 hidden inside"`

```
w = "keyword"
```

```
if w in filestr:
```

```
    indx = filestr.find(w)
```

#Load value at keyword is equal to

```
print(int(filestr[indx+len(w)+1]))
```

## Best Practice for files

- Use with statement to ensure file is closed

#Get data from file

```
fdir = "C:/dir/" + "path/to/file/"
```

```
with open(fdir + './log.txt') as f:
```

```
    filestr = f.read()
```

#File is automatically closed on leaving 'with' scope

Reading the whole file is usually efficient but for large files may need to work through line by line:

```
f.readline()      #Reads to newline "\n" and increment file pointer
```

```
f.seek(0)         #Return to the start of the file
```

- Be careful of difference in functions: readline and readlines
- In ipython, use tab to check what functions (methods) are available

## Best Practice for files

- Reading the whole file is usually efficient but for large file may need to work through line by line:

```
#Get data from file
```

```
with open('./log.txt') as f:  
    for line in f.readlines():  
        if "keyword" in line:  
            print(line)
```

- Here, only the current line is read each time so the entire file is never stored in memory

## An example with Generators (from Chris Knight)

- A generator is a function which has a persistent state, in this case the amount of the file we have read

```
#Read file in chunks
CHUNK_SIZE = 100
def yield_batch_from_file(fname, chunk_size=CHUNK_SIZE):
    with open(fname) as f:
        nlines = sum(1 for l in f)
    with open(fname) as f:
        for i in xrange(0, nlines, chunk_size):
            yield filter(None, [f.readline()
                                for j in xrange(chunk_size)])
def process_batch(batch):
    """Do some processing on a batch"""
    print(batch)

for batch in yield_batch_from_file('big_file'):
    process_batch(batch)
```

## Part 2 Summary

- Strings

```
s = "some string"
```

```
s[0:4]                                Out: some
```

```
s.title()                             Out: 'Some String'
```

```
u = s.replace("some", "a")            Out: u="a string"
```

```
s.find("m")                           Out: 2      #Index of m
```

```
"string" in s                         Out: True
```

- Use in file reading both for filepath and contents

```
fdir = "C:/dir/" + "path/to/file/"
```

```
with open(fdir + './log.txt') as f:
```

```
    filestr = f.read()
```

## Hands on session 2 – Questions


- Introduction

- Build a sentence "s" by defining and adding the 4 strings "is", "a", "this" and "sentence" in the right order (note no unique way to do these). Capitalise the first letter of each of the words. Print the first letter of each word (last one is more involved, can be skipped).

1) Write a function to add a number and a name, e.g. "filename0" from input 0 and "filename" (note str(i) converts an int to a string)

2) Download a text file (or create your own with notepad, must be plain text not word). Read the contents of the file into a string

3) Write a script to find word "keyword" inside file and get the value 4 after the = sign



text to copy to a file and read so  
you find keyword=4 hidden inside

- More Advanced

- Write a function with name of a file, read it, remove all vowels (a, e, i, o and u) and return a string made up of consonants



# **Lists and Iterators (Loops)**

## Key Concepts - Types

Define variables as one of several types

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]              # List, note square brackets tuple if ()
```

## Key Concepts - Lists

- Lists of integers

```
l = [1,2,3]
```

```
l = l + [4]    #Note l.append(4) is an alternative
```

```
Out: [1,2,3,4]
```

## Key Concepts - Lists

- Lists of integers

```
l = [1,2,3]
```

```
l.append(4)      #Note l + [4] is an alternative
```

```
Out: [1,2,3,4]
```

- We can also insert, sort and others

```
l.insert(3,0)
```

```
Out: [1,2,3,0]
```

```
l.sort()
```

```
Out: [0,1,2,3]
```

- We can index elements of the list

```
print(l[1], l[3])
```

```
Out: [1, 3]
```

Note object oriented use of a function (method). Instead of `append(l)` we have `l.append()`. The object `l` is appended to (or sorted, etc).

## Lists, Tuples and Strings

- We can also change the values in a list using indexing or sorting

```
l = [3, 2, 1]
```

```
l[1] = 4
```

```
l.sort()
```

- If you want a 'read only' list, you can use a tuple (angular brackets)

```
t = (1, 2, 3)
```

```
t[1]=4    #TypeError: 'tuple' object does not support item assignment
```

- Read only objects are called immutable, strings are immutable

```
s = "some string"
```

```
s[1] = "a" #TypeError: 'str' object does not support item assignment
```

## Lists, Tuples and Strings

- We can also change the values in a list using indexing or sorting

```
l = [3, 2, 1]
```

```
l[1] = 4
```

```
l.sort()
```

- If you want a 'read only' list, you can use a tuple (angular brackets)

```
t = (1, 2, 3)
```

```
t[1]=4    #TypeError: 'tuple' object does not support item assignment
```

- Read only objects are called immutable, strings are also immutable

```
s = "some string"
```

```
s[1] = "a" #TypeError: 'str' object does not support item assignment
```

```
l = s.split() #Convert string to a list ['some', 'string']
```

## Key Concepts - Lists

- Lists of integers

```
l = [1,2,3]
```

```
l = l + [4]
```

- But, these don't work in the same way as matrices/arrays

```
l = l + 4      TypeError: can only concatenate list (not "int") to list
```

```
l * 2  Out: [1, 2, 3, 4, 1, 2, 3, 4]
```

```
l * 2.0 Out: TypeError: can't multiply sequence by non-int of type 'float'
```

- We can make lists of any type

```
m = ["another string", 3, 3.141592653589793, [5,6]]
```

```
print(m[0], m[3][0])      #Note indexing starts from zero
```

```
Out: ("another string", 5)
```

## Loops or iterators

- Iterators – loop through the contents of a list

```
m = ["another string", 3, 3.141592653589793, [5,6]]
```

```
for item in m:
```

```
    print(type(item), " with value ", item)
```

- We could use error handling to handle operations on mixed types

```
for item in m:
```

```
    try:
```

```
        print("Rounding " , item, " to ", int(item))
```

```
    except ValueError:
```

```
        print(item, " cannot be rounded")
```



## Loops or iterators

- Iterators – loop through the contents of a list

```
m = ["another string", 3, 3.141592653589793, [5,6]]
```

```
for item in m:
```

```
    print(type(item), " with value ", item)
```

- Slightly more cumbersome for indexing

```
l = [1,2,3,4]
```

```
for i in range(len(l)):
```

```
    print("element", i, " is ", l[i] )
```

len(l) returns 4  
range(4) returns  
the list: [0,1,2,3]

## Loops or iterators

- Iterators – loop through the contents of a list

```
m = ["another string", 3, 3.141592653589793, [5,6]]
```

```
for item in m:
```

```
    print(type(item), " with value ", item)
```

- Slightly more cumbersome for indexing

```
l = [1,2,3,4]
```

```
for i in range(len(l)):
```

```
    print("element", i, " is ", l[i] )
```

- Enumerate is the Pythonic solution

```
for i, e in enumerate(l):
```

```
    print("element", i, " is ", e )
```

len(l) returns 4  
range(4) returns  
the list: [0,1,2,3]

## Loops or iterators

- Note that the following will not change the values in `l`

```
l = [1,2,3,4]
```

```
for i in l:
```

```
    i = i + 1    #Separate scope inside the for loop
```

- To add one to every element we could use

```
for i in range(len(l)):
```

```
    l[i] = l[i] + 1
```

- But list comprehension is the most Pythonic way

```
l = [i+1 for i in l]
```

equivalent  
code

```
temp = []  
for i in l:  
    temp.append(i+1)  
l = temp
```

## Iterators for files

- Iterators – more general than loops, iterate through something instead of just looping over a counter, for example lines in a file:

```
with open('file.txt') as f:  
    for line in f:  
        print(line)
```

- Letters in a word

```
for letter in "word":  
    print(letter)
```

Iterators are better than loops as they provide: A way to access the elements of an object without having to know its underlying representation. E.g. `f=open("file.txt"); f[3]` would not work

Anything which contains the `__iter__` method is iterable in Python

## Summary

- Lists store collections of data

```
l = [1, 2, 3]
```

```
m = ["another string", 3, 3.141592653589793, [5,6]]
```

- Used with iterators – more general than looping over numbers

```
for item in m:  
    print(type(item), " with value ", item)
```

- But we can still use then in this way if needed

```
l = [1,2,3]
```

```
for i in range(len(l)):  
    print(l[i])
```

## Hands on session 3 – Questions

- Introduction
  - 1) Create a list with prime numbers 7,3,5,1,2 and sort it so they are in ascending order. Use an iterator to loop through and print the output
  - 2) Write a loop to print 10 strings with names: "filename0", "filename1", ... "filename9" (note str(i) converts an int to a string)
  - 3) Using l = [1,2,3], write a loop to add a number to all elements giving [2,3,4]. Write a function to take in a list l and number N, which adds N to all elements of l and returns the list.
- More advanced
  - 1) Define two lists, one for odd and one for even numbers less than 10. Combine them to form a list of all numbers in the order [1,2,3,4,5,6,7,8,9].
  - 2) For the string s="test" and the list l = ["t","e","s","t"], we see s[0] == l[0], s[1] == l[1]. Are they equal? Can you convert the string to a list? What about list to string?
  - 3) Define a = [1,2,3]; b = a; b.append(4). Why does a = [1,2,3,4]? what about if you use b = b + [4] instead of append?

# **Dictionaries, Numpy Arrays and Classes**

## Key Concepts - Types

Define variables as one of several types

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]             # List, note square brackets tuple if ()
d = {"red":4, "blue":5}  # dictionary
```



# Dictionaries

- Dictionaries store data which can be looked up with a string

```
d = {"red":4, "blue":5}    #dictionary with red and blue keys
d["green"] = 6            #Adds an entry for green
print(d)
```

- Instead of numerical index, use a word to access elements

```
print(d["red"])    #Compare to list index with number l[3]
```

- Useful for building more complex data storage

The diagram shows a dictionary `e` with two items. The first item is `"colours" : ["red", "blue"]`. A bracket above it is labeled "item". A bracket below the key `"colours"` is labeled "key". A bracket below the value `["red", "blue"]` is labeled "Value". The second item is `"No" : [3, 6]`. A bracket above it is labeled "item". A bracket below the key `"No"` is labeled "key". A bracket below the value `[3, 6]` is labeled "Value".

```
e = {"colours" : ["red", "blue"], "No" : [3, 6]}
```

Annotations on the right side of the diagram:

- `e.items()`
- `e.keys()`
- `e.values()`

# Dictionaries

- Dictionaries store data using names

```
e = {"colours" : ["red", "blue"], "No": [3, 6]} #dictionary
```

```
e["colours"]    out: ["red", "blue"]
```

- Elements can also be accessed using key iterators

```
for key in e.keys():  
    print(key, e[key])
```

```
Out: ("colours", ["red", "blue"])  
     ("No", [3, 6])
```

- Other methods to get keys, values or items (key and value)

```
for key, value in e.items():  
    print(key, value)
```

## Dictionaries

- Could be used instead of variables, consider  $F=ma$ 
  - If we know  $F$  and  $m$ , so that  $a=F/m$

```
Newton = {}
```

```
Newton["F"] = 2.
```

```
Newton["m"] = 0.5
```

```
Newton["a"] = Newton["F"]/Newton["m"]
```

# Dictionaries

- More importantly, variables do not need to be known in advance

```
input = {}  
with open('./data.csv') as f:  
    for line in f.readlines():  
        key, value = line.split(",")  
        input[key] = float(value)
```

```
#Iterate over unknown values  
for key, value in input.items():  
    print(key, value)
```

```
#Print known input value  
print(input["Nx"])
```

```
Nsteps, 1000  
domain_x, 10.0  
domain_y, 20.0  
timestep, 0.5  
Nx, 100  
Ny, 200
```

## Summary

- Dictionaries are like lists but use strings to look up data

```
m = ["another string", 3, 3.141592653589793, [5,6]] #List
```

```
d = {"red":4, "blue":[5,6]} #dictionary
```

```
d["green"] = 8 #Can dynamically add new items
```

- List get values with a number, Dictionaries with a key

```
print(m[2], d["blue"])
```

- Iterators work with dictionaries in a similar way

```
for key, value in e.items():
```

```
    print(key, value)
```

## Key Concepts - Types

Define variables as one of several types

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]             # List, note square brackets tuple if ()
d = {"red":4, "blue":5} # dictionary
x = np.array([1,2,3])   # Numpy array
```

# Importing Numerical and Plotting Libraries

- Numpy – The basis for all other numerical packages to allow arrays instead of lists (implemented in c so more efficient)

```
import numpy as np  
x = np.array([1,2,3])
```

Dot means use array from  
module numpy.

The numpy module is just a  
big collection of Python code  
where array (and many  
other things) are defined.

Import module  
numpy and name np

Similar to:

- c++ #include
- Fortran use
- R source()
- java import (I think...)
- MATLAB adding code to path

Use tab in ipython to see what code is available (or look online)

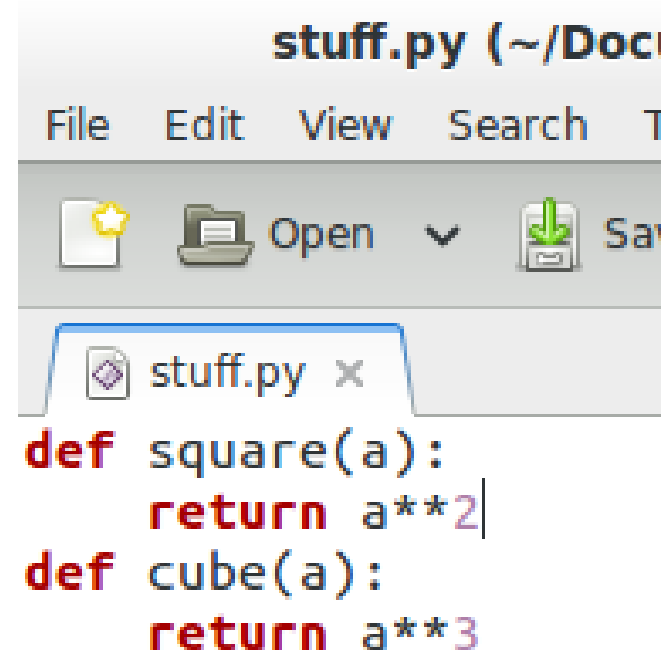
## What is a Module?

- Simply copy code to a new file, for example `stuff.py`. Any script or Python session running in the same folder can import this,

```
import stuff  
  
stuff.square(4.0)  
stuff.cube(4.0)
```

- Module code should be functions and classes ONLY. Scripts to test/run can be included using the following:

```
if __name__ == "__main__":  
    print(square(4.0), cube(4.0))
```





## Numerical and Plotting Libraries

- Numpy – The basis for all other numerical packages to allow arrays instead of lists (implemented in c so more efficient)
  - `x = np.array([1,2,3])`
  - `mean, std, linspace, sin, cos, pi, etc`
- Matplotlib – similar plotting functionality to MATLAB
  - `plot, scatter, hist, bar, contourf, imagesc (imshow), etc`
- Scipy
  - Replaces lots of the MATLAB toolboxes with optimisation, curve fitting, regression, etc. If it's not in numpy, probably in scipy
- Pandas
  - Dataframes to organise, perform statistics and plot data

NOTE: Downloading/installing packages is easier with “pip” or conda

## Numpy arrays of data

- Lists seem similar to matrices or arrays. They are not! That is why you need numpy arrays

```
import numpy as np
m = [1,2,3,4,5,6]
x = np.array(m)

#Add one to a NumPy array increments elementwise
x = x + 1    # np.array([2, 3, 4, 5, 6, 7])

#But adding one to a list will cause a TypeError
m = m + 1

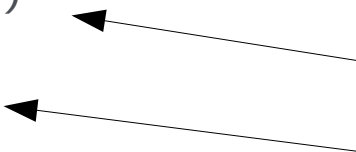
#But, conversion to numpy array if we mix types
x = x + m    #np.array([2, 4, 6, 8, 10, 12])
```

# Importing Matplotlib

- matplotlib – similar plotting functionality to MATLAB

```
import matplotlib.pyplot as plt  
x = np.array([0,1,1,2,3,5,8,13])  
plt.plot(x)  
plt.show()
```

We need the pyplot submodule of matplotlib for most things. Dot uses plot/show from matplotlib.pyplot



Use tab in ipython to see what is available (or look online)

## An Example vs MATLAB

%MATLAB

clear all

close all

x = linspace(0,2\*pi,100);

y = sin(x);

z = cos(x);

plot(x,y,'-r');

hold all

plot(x,z,'-b')

Use plot function from plt module

#python

import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(0,2\*np.pi,100)

y = np.sin(x)

z = np.cos(x)

plt.plot(x,y,"-r")

plt.plot(x,z,"-b")

plt.show()

Plotting syntax based on MATLAB

Import Plotting module  
matplotlib as plt

## An Example vs MATLAB

%MATLAB

```
clear all
```

```
close all
```

```
x = linspace(0,2*pi,100);
```

```
y = sin(x);
```

```
z = cos(x);
```

```
plot(x,y,'-r');
```

```
hold all
```

```
plot(x,z,'-b')
```

plot function has been imported

#python

```
from numpy import *
```

```
from matplotlib.pyplot import *
```

Import all

```
x = linspace(0,2*pi,100)
```

```
y = sin(x)
```

```
z = cos(x)
```

```
plot(x,y,"-r")
```

```
plot(x,z,"-b")
```

```
show()
```

Better not to do this to  
avoid nameclashes

## Key Concepts - Types

- Show how to use the command prompt to quickly learn Python
- Introduce a range of data types (Note everything is an object)

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]             # List, note square brackets tuple if ( )
d = {"red":4, "blue":5}  # dictionary
x = np.array([1,2,3])    # Numpy array
```

- Show how to use them in other constructs including conditionals (**if** statements) iterators (**for** loops) and functions (**def** name)
- Introduce external libraries numpy and matplotlib for scientific computing

## Key Concepts - Types

- Show how to use the command prompt to quickly learn Python
- Introduce a range of data types (**Note everything is an object**)

```
a = 3.141592653589      # Float
i = 3                   # Integer
s = "some string"       # String
l = [1,2,3]             # List, note square brackets tuple if ()
d = {"red":4, "blue":5}  # dictionary
x = np.array([1,2,3])    # Numpy array
```

- Show how to use them in other constructs including conditionals (**if** statements) iterators (**for** loops) and functions (**def** name)
- Introduce external libraries numpy and matplotlib for scientific computing

## What is an Object?

- Three types of programming we've used so far...
  - Procedural
  - Functional
  - Object oriented
- Everything in Python is an object
  - You can get away with procedural or functional programming
  - Worth understanding a little bit about objects
  - Slightly different way of thinking



# Classes in Python

- A person class could include name, age and method say their name

```
class Person():
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```


```
        self.age = age
```

```
    def say_name(self):
```

```
        print("Hello, I'm "
```

```
              + self.name)
```

Python provides the following syntax for a constructor, a function which **MUST** be called when creating an instance of a class



# Classes in Python

- A person class could include name, age and method say their name

```
class Person():
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def say_name(self):
```

```
        print("Hello, I'm "  
              + self.name)
```

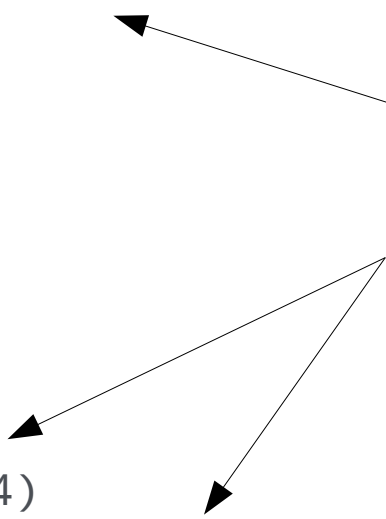
```
bob = Person('Bob Jones', 24)
```

```
jane = Person('Jane Bones', 32)
```

```
bob.say_name()
```

```
jane.say_name()
```

Python provides the following syntax for a constructor, a function which **MUST** be called when creating an instance of a class  
Called automatically when we instantiate



## We have already seen this...

- String is an object with associated methods

```
s = "some string"
```

```
s.title()                                Out: 'Some String'
```

```
s.find("o")                             Out: 1
```

```
t = s.replace("some", "a")              Out: t="a string"
```

- and for lists

```
l.append(4)      #Note l + [4] is an alternative
```

```
l.sort()
```

- Or, in fact, anything else in Python

```
a = 3.5
```

```
a.as_integer_ratio()      #Output (7, 2)
```

- In ipython, use tab to check what functions (methods) are available

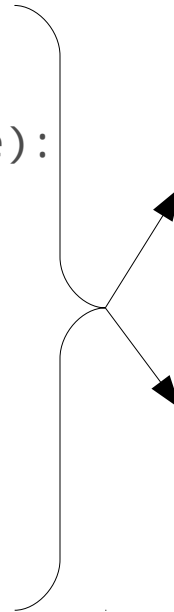
# Classes in Python

- A person can train in a particular area and gain specialist skills

```
class Person():  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def say_name(self):  
        print("Hello, I'm "  
              + self.name)
```

```
class Scientist(Person):  
    def do_science(self):  
        print(self.name +  
              'is researching')
```

```
class Artist(Person):  
    def do_art(self):  
        print(self.name +  
              'is painting')
```



# Classes in Python

- A person can train in a particular area and gain specialist skills

```
class Person():  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def say_name(self):  
        print("Hello, I'm "  
              + self.name)  
  
class Scientist(Person):  
    def do_science(self):  
        print(self.name +  
              'is researching')  
  
class Artist(Person):  
    def do_art(self):  
        print(self.name +  
              'is painting')
```

```
graph TD; Scientist --> Person; Artist --> Person;
```

```
bob = Artist('Bob Jones', 24)  
jane = Scientist('Jane Bones', 32)  
bob.say_name(); bob.do_art()  
jane.say_name(); jane.do_science()
```

# Classes in Python

- A number class which includes methods to get square and cube

```
class Number():
```

```
    def __init__(self, a):  
        self.a = a
```

```
    def square(self):  
        return self.a**2
```

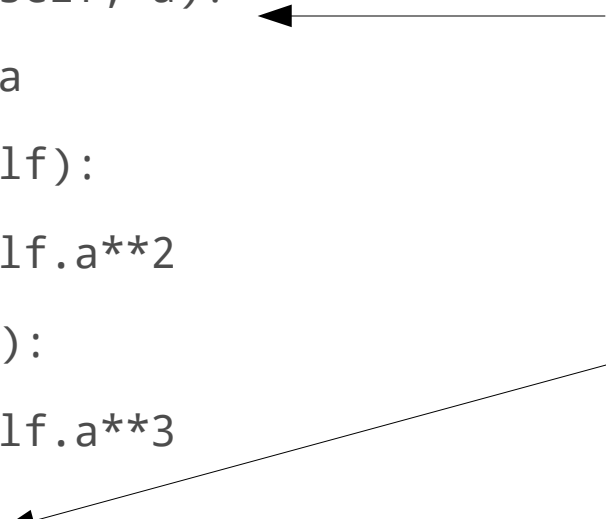
```
    def cube(self):  
        return self.a**3
```

```
n = Number(4.5)
```

```
n.square()           #Out: 20.25
```

```
n.cube()             #Out: 91.125
```

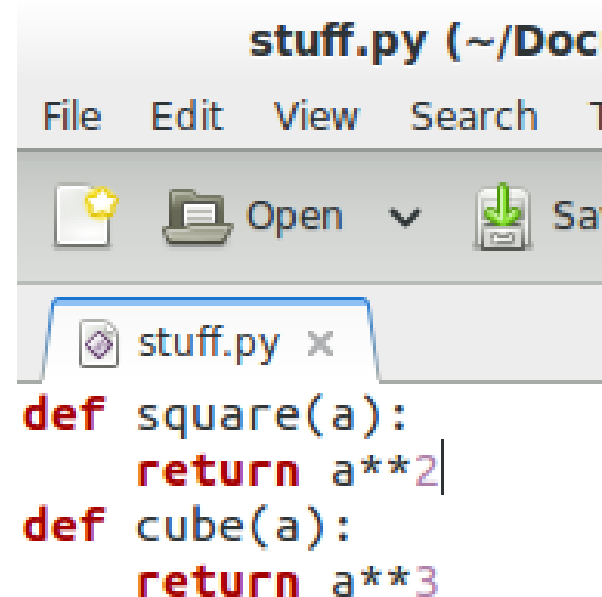
Python provides the following syntax for a constructor, a function which **MUST** be called when creating an instance of a class. Called automatically when we "instantiate"



# Classes vs Modules

- What is the difference between classes and code in a module?

```
class Number():  
    def __init__(self, a):  
        self.a = a  
    def square(self):  
        return self.a**2  
    def cube(self):  
        return self.a**3
```



- A module, e.g. numpy, is a collection of functions and classes you can access using the module name followed by a dot, e.g. "numpy.stuff".
- A class is used to instantiate objects.

## Hands-On Session 4

### Introductory Questions

- 1) Dictionary – Create a dictionary `shape_sides` with keys "triangle", "square" and "pentagon" and values 3, 4 and 5 respectively. Iterate and print all items
- 2) Numpy arrays – Import the numpy module, create an numpy arrays of values from 1 to 5 and add one to each entry.
- 3) Classes – Create a class called `number` which takes an input `x` in its constructor and stores it (`self.x = x`). Add a method to square the (`self.x`) value and return
- 4) Create a module containing a function which adds two numbers `a` and `b`, returning their sum. import into a script and print output

### More Advanced Questions

- 5) Read the input file to the right and store the results inside a dictionary with keys "integer", "float" and "string" and values stores with appropriate type

```
integer; 10  
float; 1.25  
string; "hello"
```



# Summary

## Evolutions of a Python Project

- Python session to try things, copy to a simple script and test
- Group repeated code into functions to avoid repetition:
  - Reduces potential errors as less code to check
  - Improves readability as clear modular parts with a clear interface **which can be tested**
  - Easier to maintain and less to change as design evolves
- Collect together similar functions in a module
- Group functions acting on an object into a class for that object
- Utilise inheritance to further reduce code volume
- Create a package by adding `__init__.py` file to folder
- Add test scripts to packages

## Version Control

- Once you have some code, put it into a code repository
  - Backup in case you lose your computer
  - Access to code from home, work and anywhere else.
  - Allows you to keep a clear history of code changes
  - Only reasonable option when working together on a code
- Three main repositories are git, mercurial and subversion.
- Most common is git, a steep learning curve and helps the maintainer more than the developer (in my opinion). Mercurial may be better... Subversion is often disregarded due to centralised model.
- Range of free services for hosting, Imperial has a paid github account <https://github.com/> so you can host close source projects

## Summary

- Background and motivations for this course
  - MATLAB is the main programming language taught at Imperial
  - Python provides similar plotting, numerical analysis and more
- Some key concepts
  - Data types, lists/arrays, conditionals, iterators and functions
  - Modules for scientific computing: numpy and matplotlib
  - Clean syntax, ease of use but no checking!
- Advantages of learning Python
  - General programming (better both in academia and outside)
  - Allows an integrated framework and can be bundled with code
  - Open source libraries with tutorials and excellent help online

## What to do next?

- Find a project
  - Use Python instead of your desktop calculator
  - Ideally something at work and outside
- Use search engines for help, Python is ubiquitous - often you can find sample code and tutorials for exactly your problem
  - Stackoverflow is usually the best source of explanation
  - Official documentation is okay as a reference but not introductory, look for many excellent tutorials, guides and videos
  - `help(function)` in python. Tab, `?` or `??` in ipython
- Be prepared for initial frustration!
  - Worth the effort to learn

## Other libraries

- Graphical User Interfaces (GUI) e.g. Tkinter, wxpython, pyGTK, pyQT
- Multi-threading and parallel e.g. Subprocess, mpi4py
- Image and video manipulation e.g. pyCV, PIL
- Machine learning e.g. Scikit-learn, Pybrain
- Differential equations solvers e.g. FEniCS, Firedrake
- Databasing and file storage e.g. h5py, pysqlite
- Web and networking e.g. HTTPLib2, twisted, django, flask
- Web scraping – e.g. scrapy, beautiful soup
- Any many others, e.g. PyGame, maps, audio, cryptography, etc, etc
- Wrappers/Glue for accelerated code e.g. LAMMPS, OpenFOAM, HOOMD, PyFR (CUDA), etc
- It is also easy to create your own packages

## Plan for Tomorrow

- Introduction – why learn Python and review of today
- Dictionaries, Numpy arrays and classes
- More on Numpy and matplotlib
- Loading data from files
- Using Python as glue
- More advanced plotting
- A complete post-processing example
- Best practice and summary

## What to do next?

- We will continue tomorrow... If you can't join us, and we didn't cover something you needed for your work, please ask. I will also send notes to everyone who signed up.
- Please provide feedback on today
  - Was the course useful? What could be improved?
  - I believe Python should be taught at undergraduate level here at Imperial. Please support this by filling in the questionnaire, I will present the results at Wednesday's HPC session (from 10 – 12)

<http://bit.ly/2yf1vka>

Or Link can be found here:

[http://cpl-library.org/python\\_feedback.shtml](http://cpl-library.org/python_feedback.shtml)