

# Computer Programming in Python

## A Primer for 4C16

Hugh Denman

Trinity College, Dublin

`hdenman@tcd.ie`

September 25, 2017

# A primer for engineers

- For engineers
  - ▶ not principally about the programming language
  - ▶ more about exploiting the computer as a tool
  - ▶ (in contrast with e.g. web development)
  - ▶ technology stack; interaction of layers; vertical understanding
- Constraints, context, contrasts

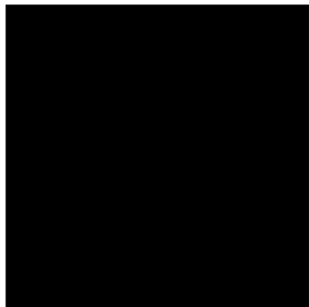
# Overview

- 1 The Computer
- 2 Programming
- 3 Python
- 4 The 4C16 Platform – SCM
- 5 The 4C16 Platform – Labs

# Section 1

## The Computer

# Essential components



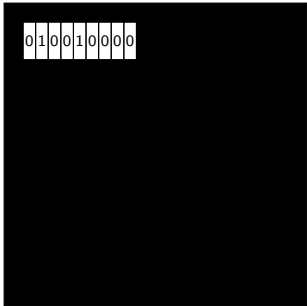
- Computer as black box

# Essential components



- Register: stores a value

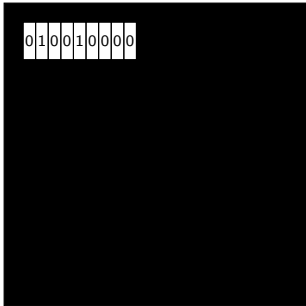
# Essential components



010010000

- Register: stores a value
- Value stored as binary digits

# Essential components

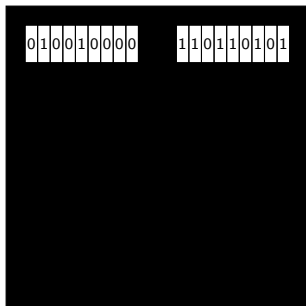


010010000

- Register: stores a value
- Value stored as binary digits
- Fixed size: e.g. 32 or 64 binary digits
- Register size in bits defines 'word size'
- Word = native, efficient value size for processor

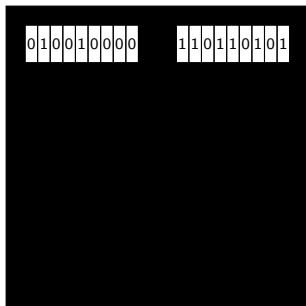


# Essential components



- Register: stores a value
- Value stored as binary digits
- Fixed size: e.g. 32 or 64 binary digits
- Register size in bits defines 'word size'
- Word = native, efficient value size for processor
- — minimum 2 registers to be useful

# Essential components



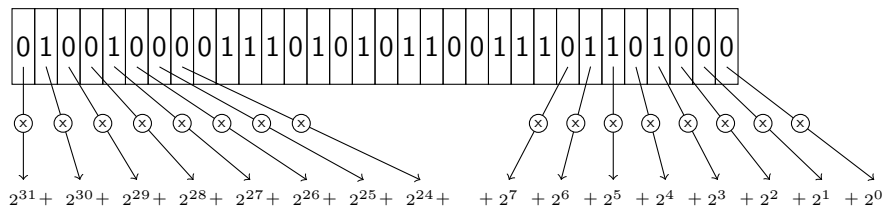
- Register: stores a value
- Value stored as binary digits
- Fixed size: e.g. 32 or 64 binary digits
- Register size in bits defines 'word size'
- Word = native, efficient value size for processor
- — minimum 2 registers to be useful
- The value represented by a word is extrinsic

# Values in the Computer

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Interpretation of values

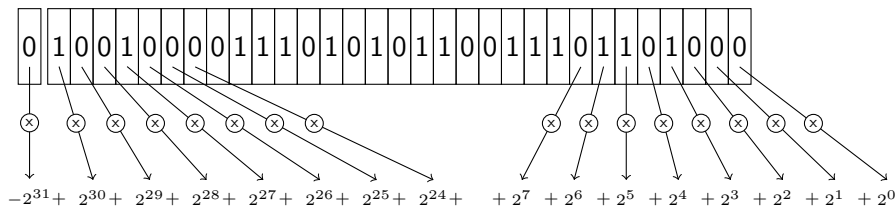
# Values in the Computer



## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)

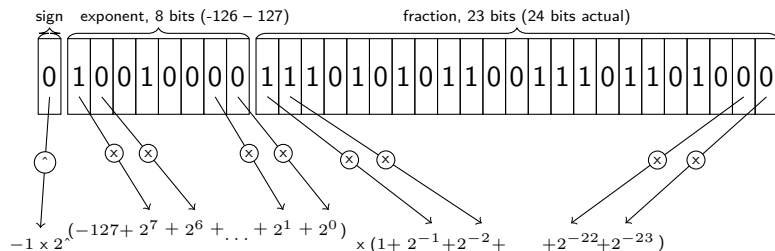
# Values in the Computer



## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)
- 1 32-bit integer: 1215653736 (signed: two's complement)

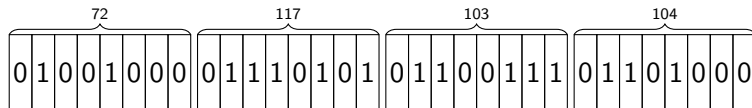
# Values in the Computer



## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)
- 1 32-bit integer: 1215653736 (signed: two's complement)
- 1 32-bit float: 251293.625

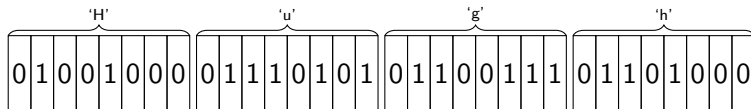
# Values in the Computer



## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)
- 1 32-bit integer: 1215653736 (signed: two's complement)
- 1 32-bit float: 251293.625
- 4 8-bit integers: 72, 117, 103, 104

# Values in the Computer

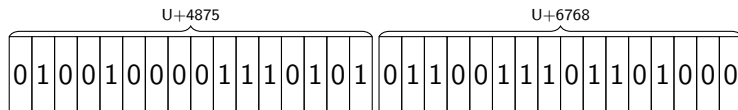


## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)
- 1 32-bit integer: 1215653736 (signed: two's complement)
- 1 32-bit float: 251293.625
- 4 8-bit integers: 72, 117, 103, 104
- 4 ASCII-encoded symbols (letters): "Hugh"



# Values in the Computer



## Interpretation of values

- 1 32-bit integer: 1215653736 (unsigned)
- 1 32-bit integer: 1215653736 (signed: two's complement)
- 1 32-bit float: 251293.625
- 4 8-bit integers: 72, 117, 103, 104
- 4 ASCII-encoded symbols (letters): "Hugh"
- 2 Unicode UTF16-encoded symbols (letters): 轆杨

## Essential components (continued)



Registers

## Essential components (continued)



- add / multiply / increment / compare
- (up to) billions of times a second
- only very simple operations
- instructions are specific to value type

## Essential components (continued)



- Get one / a few number(s) from memory
- Memory = a vast sea of words
- Very slow compared to ALU

## Essential components (continued)



- Fetch & decode instructions from memory
- Branching / conditionals
- I/O ports (talking to specialized hardware)
- Interrupts

# Section 2

## Programming

# Kinds of Programming

- Scientific computing
  - ▶ Calculation of values for scientists / engineers
  - ▶ Numerically intensive, performance-critical
  - ▶ Our present concern
- Systems programming
  - ▶ as in Operating System
  - ▶ also programming languages (compilers / interpreters)
  - ▶ Production of abstractions, e.g. memory management, on-disk files
  - ▶ Target audience: programmers
- Application programming
  - ▶ Production of applications, for word processing, video & music production, spreadsheets, etc.
  - ▶ Also browsers / websites / YouTube
  - ▶ Not numerically intensive, design-critical, computer mostly idle
  - ▶ Target audience: end users

# Programming Languages

Programming languages provide, in varying degrees:

- data types, type safety (correctness constraints)
- libraries of useful functions
- abstractions
- error checking



# Programming Languages

## Assembly language(s)

- Transcribed (rather than translated) to native machine code

```
1  section .data
2  x: dd 10 ; 32-bit word initialized to 10
3          ; x is a label for a memory location
4  section .text
5  main:
6  mov eax,[x]
7  cld      ; message of input before division
8  mov ecx, 3
9  idivl    ; signed rather unsigned division
10 mov [x], eax
```

<https://www.csee.umbc.edu/portal/help/nasm/sample.shtml>

# Programming Languages

Compiled languages, such as C / C++

- Program compiled (translated) to machine code (one-time operation)
- Programmer provides explicit information (declarations)
- Predictable performance

```
1 // Declare that x is a signed integer, initialized to 10
2 int x = 10;
3 // Correct instructions for signed automatically issued
4 x = x / 3;
```

# Programming Languages

## Scripting languages (Python, JavaScript, Ruby)

- Program interpreted as it runs (every time)
- Programmer only needs to provide indication of intent; lots of inference and assumptions at work in language
- Less-predictable performance (housekeeping)

```
1 x = 10 # No declaration!  
2 x = x / 3 # anything could happen
```

# Why Python

Highly & increasingly popular amongst machine learning practitioners, as well in general.

Python offers the convenience of high-level facilities for general programming, combined with specialized high-performance libraries (which are not themselves written in Python) for number crunching.

Also:

- Free
- Reasonably friendly
- Reasonably strict
- Can be nearly industrial
- (Alas?) Not Matlab

Not a good language for implementing high-performance cutting-edge algorithms, but a good language for using them.

# Why Python

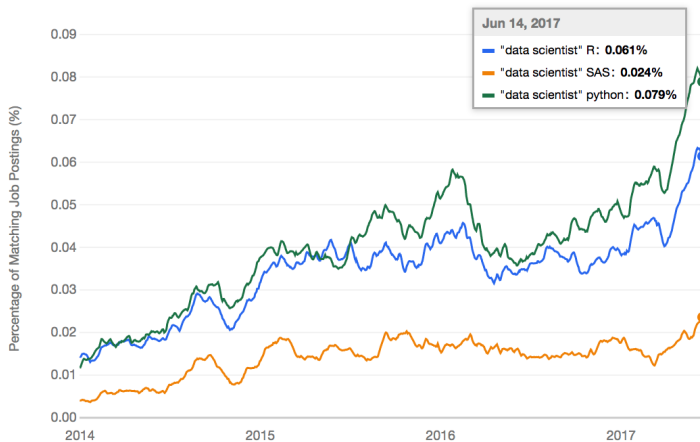
Worldwide, Sept 2017 compared to a year ago:

| Rank | Change | Language    | Share  | Trend  |
|------|--------|-------------|--------|--------|
| 1    |        | Java        | 22.4 % | -0.8 % |
| 2    |        | Python      | 17.0 % | +4.0 % |
| 3    |        | PHP         | 8.7 %  | -1.0 % |
| 4    |        | C#          | 8.1 %  | -0.4 % |
| 5    |        | Javascript  | 8.0 %  | +0.6 % |
| 6    |        | C++         | 6.8 %  | -0.2 % |
| 7    |        | C           | 6.1 %  | -1.1 % |
| 8    | ↑      | R           | 3.7 %  | +0.6 % |
| 9    | ↓      | Objective-C | 3.5 %  | -1.4 % |
| 10   |        | Swift       | 2.9 %  | -0.1 % |

Source: Popularity of Programming Languages project.

<http://pypl.github.io/>.

# Why Python



Source: Indeed.com job trends.

<https://www.indeed.com/jobtrends/q-%22data-scientist%22->

[R-q-%22data-scientist%22-SAS-q-%22data-scientist%22-python.html](https://www.indeed.com/jobtrends/q-%22data-scientist%22-SAS-q-%22data-scientist%22-python.html).

# Section 3

## Python

# Running Python

- Experiment interactively with a Read / Evaluate / Print Loop (REPL)
  - ▶ Online: <https://repl.it/languages/python3>
  - ▶ On your computer: `python3` in Terminal (OS X)
- Develop code to be kept and reused in a text file
- Exploratory data analysis: Use a workbook environment (e.g. Jupyter)



# Running Python: log in to 4C16 system

- Open a new browser window (ctrl-N)
- Go to `http://turing.mee.tcd.ie/`
- Login with your username and password

Change your password! The link is at the bottom right:

[Change your password](#) | [Restart Tunnel](#) | [Sign Out](#)

This system manages a remote instance for you; you can start it from the web dashboard:

## 4C16 Dashboard: Hugh Cluster

Your cluster is hugh-t-t; it is presently down.

Start Cluster

- Press the 'Start Cluster' button
- It should take 30–40 seconds

From the dashboard, you can launch a terminal, an editor, or the Jupyter environment.

# 4C16 Dashboard: Hugh Cluster

Your cluster is hugh-t-t; it is presently up.

[Open your Terminal](#)

[Open your Editor](#)

[Open your Jupyter Notebook](#)

Stop Cluster

# Running Python: Start a Terminal

- Click on the 'Open your Terminal' link

The tab that appears is a Unix Terminal (an interface for relaying text between a computer and a screen / keyboard). This terminal is running a shell, which interprets text commands for the computer to run.

The particular shell we're using is called bash, and the computer is running the Linux operating system.

- Type in 'python -V' (capital V) and press Enter (python -V↵).
  - ▶ The system will respond by telling you what version of the Python programming language is available — it should be 'Python 3.6'.
- Type in 'python' and press Enter (python↵).
  - ▶ Now you are running a Python REPL inside the terminal.
  - ▶ You can type `exit()`↵ or press Ctrl-D to return to the shell.

# Python Statements

- Programming language *statement*  $\sim$  sentence
- Building blocks with which we describe what we intend for computer
- Simplest kind of statement is an *expression*: computes a value
- All values have a *type*
- Another common kind of statement is an *assignment*: associates a value with a name (variable)
- Generally in Python, a statement is exactly one line (i.e. runs to end of line)

# Expression statements

```
1 Python 3.5.1 (default, Apr 18 2016, 03:49:24)
2 >>> 5
3 5
4 >>> 1.23
5 1.23
6 >>> -3 + 0.4j
7 (-3+0.4j)
8 >>> "Hugh"
9 'Hugh'
10 >>> x
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 NameError: name 'x' is not defined
```

# Expression statements with operators

```
1 >>> 5 + 3
2 8
3 >>> 1.23 - 2 * (2 + 3/9)
4 -3.4366666666666667
5 >>> -3 + 0.4j
6 (-3+0.4j)
7 >>> "Hugh" + " " + 'Denman'
8 'Hugh Denman'
9 >>> 10 / 3
10 3.3333333333333335
11 >>> 10 // 3
12 3
13 >>> 0.2 + 0.1
14 0.30000000000000004
```



# Value types

```
1 >>> type(3)
2 <class 'int'>
3 >>> type(3.3)
4 <class 'float'>
5 >>> type(3.3 + 3j)
6 <class 'complex'>
7 >>> type('hugh' + 'harry')
8 <class 'str'>
9
10 >>> print(print('hugh'))
11 hugh
12 None
13 >>> type(print('hugh'))
14 hugh
15 <class 'NoneType'>
```

# Assignment statements and variables

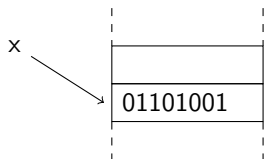
```
1 >>> x = 3          # no need to declare variables
2 >>> x + 5
3 8
4
5 >>> eigenvalue = 2 # ... which can have pitfalls
6 >>> eignvalue = eigenvalue + 1
7
8 >>> type(x = 3)     # assignment is not an expression
9 TypeError: type() takes 1 or 3 arguments
10
11 >>> type(x)         # variables may change type, or:
12 <class 'int'>       # 'python is dynamically typed'
13 >>> x = 3.3
14 >>> type(x)
15 <class 'float'>
```

# Python is dynamically typed

- Assembly language is untyped: variable is just bit pattern; no restrictions
- C / C++ / Java: statically typed: variable has fixed, declared type
- Python (Ruby, Javascript): variable type can vary, must be checked with every access

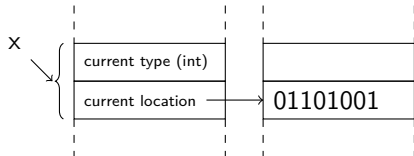
C / C++

```
1 int x = 0;  
2 x = x + 3;
```



Python / Ruby / etc.

```
1 x = 0  
2 x = x + 3
```



# Statement parsing

```
1 >>> x = 3      # A newline marks the end of a statement
2 >>> x + 5      # Trailing space is ok
3 8
4 >>> x +        # The line must make a complete statement
5 SyntaxError: invalid syntax
6 >>> (x +       # Unless you use parentheses
7 ... 3)
8 4
9 >>> y = 1; y + 3; y + 1
10 4             # A semicolon also terminates a statement
11 2
12
```

# Python functions

A function definition permits the assigning of a name to a statement or group of statements, just as a variable definition permits the assigning of a name to a value.

User-defined functions provide a higher-level abstraction / semantic unit. This is critical for readable, maintainable programs.

```
1 >>> def say_stuff(): # defining a function
2 ...     print("some day") # four spaces at the start
3 ...     print("when the world is cold")
4 ...     print("i will feel a glow")
5 ...     print("just thinking of you")
6 ...     print("and the way you look tonight")
7 ...     # newline (press return), no spaces
8 >>> say_stuff() # calling / invoking a function
9 some day <etc...>
```

# Python functions: writing in editor

Open the editor:

- Click on the 'dashboard' tab (the first one which opened)
- Click on 'open your editor'
  - ▶ A new tab will appear with an editor (strictly speaking this is the ACE editor running inside the Codiad IDE).

Add a file containing a new function:

- Right click on 'code'
- Choose 'new file' (call it 'test.py')
- Put the function 'say\_stuff' into this file (previous slide)
- Save it (ctrl-S)

## Python functions: running from file

To run this function from the file, we must first import it into python:

- Go back to the Python tab

```
1 >>> import test
2 >>> test.say_stuff()
```

Notice that the function is within the 'test' namespace.

Functions can also be imported into the global namespace:

```
1 >>> from test import say_stuff
2 >>> say_stuff()
```

## Python functions: reloading after a change

If you want to test an updated version of the function, you can either exit python and restart, or reload within the session:

- Make a change to the function in the editor and save

```
1 >>> import importlib          # enables reloading
2 >>> test = importlib.reload(test) # actually reload
3 >>> test.say_stuff()           # Run updated version
```



# Python function parameters

A function can take parameters, which are variables scoped to the definition of the function. When the function is called, or invoked, we supply an expression which provides the value of the parameter variable. Writing a function which takes parameters is always better than copy-pasting a block of code and updating the values used in the duplicate block, as it results in more readable, maintainable programs.

```
1 >>> def say_hello(x): # Function takes parameter 'x'
2 ...     print("Hello, " + x + "!")
3 ...
4 >>> say_hello("Springfield")
5 Hello, Springfield! # Called with argument "Springfield"
```

# Why use functions

- Label a conceptual block
- Decompose problem into layers
- Parameterize a computation

Even very simple functions are preferable to using a raw expression, for readability and maintainability.

```
1 >>> def fahrenheit_to_celcius(x):  
2 ...     return (x - 32) * 5 / 9  
3 ...  
4 >>> print fahrenheit_to_celcius(82)  
5 27.77777777777778  
6
```

# Boolean values and control flow

A boolean value is either True or False.

Booleans are used principally for if or while statements.

```
1 >>> x = False    # Boolean literal
2 >>> it_is_raining = (current_precipitation() == 'rain')
3 >>> if (it_is_raining): # using a boolean in a conditional
4 ...     deploy_umbrella()
5 ... else:
6 ...     don_t_shirt()
7
8 >>> while(current_precipitation() == 'rain'):
9 ...     stare_out_window()
10 >>> go_outside()
```

All values can be converted to / evaluated as Boolean and used in conditionals. All values are True except None, any number evaluating to 0, and any empty collection (including an empty string).

# Compound types

Compound types contain multiple simple types.

```
1 >>> l = ['a', 3, 8.3, [4, 4]] # list
2 >>> print(l[0])                # element access by index
3 a
4 >>> print(l[1:3])              # slice access by range
5 [3, 8.3]
6 >>> t = ('Hugh', 1978, ['Francis', 'Norah-Mae']) # tuple
7 >>> t[0]
8 'Hugh'
9
10 >>> d = {'dog': 'mammal',      # dictionary
11 ...     'crocodile': 'reptile'}
12 >>> d['dog']
13 'mammal'
```

The members of a compound value need not be of the same type.

## The value of a compound type is its location

```
1 >>> def add_one(x):
2 ...     x = x + 1
3 ...
4 >>> a = 2; add_one(a); print(a)
5 2
6 >>> def add_one_to_first(x):
7 ...     x[0] = x[0] + 1
8 ...
9 >>> l = [2, 'hugh']; add_one_to_first(l); print(l)
10 [3, 'hugh']
11
12 >>> add_one(l);
13 TypeError: can only concatenate list (not "int") to list
14 >>> add_one_to_first(a)
15 TypeError: 'int' object is not subscriptable
```

# Loops and Lists

Transforming a list:

```
1 >>> my_list = [5, 6, 7, 8, 9, 10]
2 >>> my_list = list(range(5,11))    # (list contents identical)
3
4 >>> new_list = []
5 >>> for i in range(len(my_list)):    # Traditional
6 ...     new_list.append(my_list[i] * 2)
7 ...
8 >>> new_list = [2*x for x in my_list] # More Pythonic
```

# Loops and Lists

Summing (reducing) a list:

```
1 >>> my_sum = 0
2 >>> for i in range(len(my_list)):      # C-thinking
3 ...     my_sum = my_sum + my_list[i]
4 ...
5 >>> my_sum = 0
6 >>> for e in my_list:                  # More pythonic
7 ...     my_sum = my_sum + e
8 ...
9     # (as it happens there is a built-in 'sum' function)
10 >>> my_sum = sum(my_list)
```

# Modules

Python comes with a large library of useful functions. These are organized into *modules*.

```
1 >>> import math
2 >>> print(math.sqrt(2))
3 1.4142135623730951
4
5 >>> [0.1] * 10
6 [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
7 >>> print(sum([0.1] * 10))
8 0.9999999999999999
9 >>> from math import fsum
10 >>> print(fsum([0.1] * 10))
11 1.0
```



# Common & Useful Modules

Built-in modules (the Python Standard Library):

- Interacting with host operating system and computer
- Internet communications
- Reading and writing simple sound and image formats

Third-party modules

- More complex sound and image formats
- Fast numerical processing and scientific computation
- Graphing and plotting data

A Python programmer will typically organize their own code into several modules when writing larger programs.

# NumPy

```
1 >>> import random as rand
2 >>> import timeit as ti
3 >>> d = [int(1000*rand.random()) for i in range(10000)]
4 >>> num_reps = 10000
5 >>> p_t = ti.timeit('[abs(x) for x in data]',
6 ... setup="from __main__ import data", number=num_reps)
7 >>> import numpy as np
8 >>> d_n = np.asarray(data)
9 >>> np_t = ti.timeit('np.abs(data_for_numpy)',
10 ... setup="import numpy as np; from __main__ import d_n",
11 ... number=num_reps)
12 >>> print("Python: " + str(p_t) +
13 ... "; NumPy: " + str(np_t) +
14 ... "; Ratio: " + str(p_t / np_t))
15 Python: 6.4231; NumPy: 0.0849; Ratio: 75.6106
```

# Not covered

- Other types (sets, fractions, ...)
- Classes / object orientation
- Error handling & exceptions
- File I/O

## Going further

When looking for Python help online, be sure to use the search term `python3`, as several important things have changed from the previous, still very widespread, version 2.

- `print` is a function
- `xrange` is now `range`

## Section 4

### The 4C16 Platform – SCM

# Git overview

It is essential to use source code management (version control) for any serious programming.

We use the Git SCM system—but we will only use some features.

- Initialise a *repository* (or *repo*)
- Make some modifications to your code
- Review modifications & decide which to *stage*
- *Commit* staged modifications
- *Push* to a remote repo (back up / collaborate)

Your cluster is ephemeral: always commit & push!

# Git workflow

When you open your terminal, you are automatically in a directory (folder) called 'code'. This is an empty git repository. Use the command `git status` to check the state of the repo:

```
1 tcd@instance:~/code$ git status
2 On branch master
3 No commits yet
4 Untracked files:
5
6   test.py
7
8 nothing to commit but untracked files present
```

## Git workflow

The git workflow is to prepare a copy, or image, of your work that represents a step forward, and then *commit* that copy to the record.

```
1 tcd@instance:~/code$ git add test.py
2 tcd@instance:~/code$ git status
3 On branch master
4 No commits yet
5 Changes to be committed:
6   (use "git rm --cached <file>..." to unstage)
7       new file:   test.py
8 tcd@instance:~/code$ git commit -m "Added test file."
9 [master (root-commit) 5f7fe1e] Added test file.
10    1 file changed, 1 insertion(+)
11    create mode 100644 test.py
```

Here we have added the file `test.py` to the image for committing, checked that the repository is in the state we expect (via `git status`), and then committed our change.



# Git workflow

After modifying the file 'test.py' in the editor:

```
1 tcd@instance:~/code$ git status
2 On branch master
3 Your branch is ahead of 'origin/master' by 1 commit.
4   (use "git push" to publish your local commits)
5 Changes not staged for commit:
6   (use "git add <file>..." to update what will be committed)
7   (use "git checkout -- <file>..." to discard changes)
8 modified:   test.py
```

# Git workflow

Examine the modification using `git diff`.

```
1 tcd@instance:~/code$ git diff
2 diff --git a/test.py b/test.py
3 index 9e9c9f1..aa7714e 100644
4 --- a/test.py
5 +++ b/test.py
6 @@ -1,2 +1,2 @@
7   def say_help():
8   -     print("Help")
9   \ No newline at end of file
10  +     print("Help me out")
```

## Git workflow

Commit all modifications with `git commit -a -m "<your message here>":`

```
1 tcd@instance:~/code$ git commit -a -m "Updated help message."  
2 [master 63db699] Updated message.  
3 1 file changed, 1 insertion(+), 1 deletion(-)
```

Update the remote (permanent backup) copy of your repo with `git push`:

```
1 tcd@instance:~/code$ git push  
2 Counting objects: 6, done.  
3 Compressing objects: 100% (2/2), done.  
4 Writing objects: 100% (6/6), 518 bytes | 172.00 KiB/s, done.  
5 Total 6 (delta 0), reused 0 (delta 0)  
6 To ssh://localhost:2000/hugh.denman-code  
7    5f7fe1e..63db699  master -> master
```

## Section 5

### The 4C16 Platform – Labs

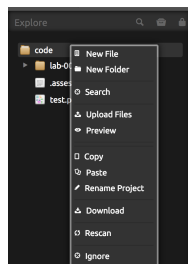
# The Lab System

Git-based, automatic assessment.

Check out skeleton files for lab 0 (these contain the assignment):

```
1 tcd@instance:~/code$ setup-lab 0
```

In the Editor tab, right click on 'code' and choose 'rescan' to view the newly added files.



Send files to server to be checked:

```
1 tcd@instance:~/code$ submit-lab 0
```

# The Lab System

Remember: you can view the progress you've made with

```
1 tcd@instance:~/code$ git status
2 tcd@instance:~/code$ git diff
```

If you decide that you've messed up, you can go back to your last commit with:

```
1 tcd@instance:~/code$ git checkout .
```

It's a good idea, therefore, to test and commit frequently.

## Lab 0 Exercise 1

- Double-click on 'lab-00' in the editor
- Double-click on 'ex\_1.py' (exercise 1)

This first exercise has three parts. Read the text to get a feel for the problems. Go to the terminal tab and enter:

```
1 python lab-00/ex_1.py
```

This will try to execute the code appearing in the file ex\_1.py under

```
1 if __name__ == "__main__":
```

You will get errors at first as the lab has not been completed.

## Lab 0 Exercise 1

Work through the three parts of exercise 1. Save the file and run it in the terminal as you complete each part, to check your progress.

```
1 python lab-00/ex_1.py
```

- Uncomment and fix up the 'safe\_div' function (indents, colons).
- Fix the error in the function 'compute\_sum'
- Add a new function 'compute\_product' to compute the product of three numbers.

You can send the lab for assessment whenever you want, but you must commit your changes to git first:

```
1 git commit -a -m "Finishing exercise 1"  
2 submit-lab 0
```



## Lab 0 Exercise 2

Open the file `ex_2.py` in the editor and have a look. The task here is to flesh out an implementation of Newton's method (Newton Raphson) for finding a root of a function  $f(x)$  (a value  $\hat{x}$  such that  $f(\hat{x}) = 0$ ).

The method is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- Fix the indents.
- Use `if derivative == 0:` to check if the search has to be abandoned.
- Use the `abs()` function to check the size of the update.
- Use `if i > 100:` to check the number of iterations

To check:

```
1 python lab-00/ex_2.py
2 git commit -a -m "Finishing exercise 2"
3 submit-lab 0
```

Jupyter offers:

- Workflow via cells
- In-line plots
- Automatic checkpointing

Great for exploratory data analysis, interacting with models, visualizing data.

# Jupyter

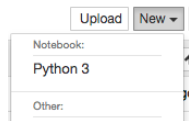
Before we start Jupyter, we will create a function to generate some data to visualize.

Use the Codiad editor (not Jupyter) to create a file in the lab-00 directory. Call it `my_jupyter.py`, and enter this:

```
1 import numpy as np
2 def gen_data():
3     # Numpy array, ranging from 0--25 in steps of 0.1:
4     x = np.arange(0, 25, 0.1);
5     # y as a function of x:
6     y = np.sin(x)
7     return (x, y)
```

# Getting started with Jupyter

- Click 'Open your Jupyter Notebook'
- In the Jupyter tab, click on 'lab-00'
- Then click 'New', and choose 'Notebook | Python 3'
- In the new notebook, click on 'Untitled' (up at the top), and enter a new name, e.g. 'lab-0-test'.



# Jupyter notebook setup

Enter this into the top cell and press 'ctrl+enter' to evaluate it.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import importlib
4 %matplotlib inline
5
```

This is important setup code which will ensure that your Jupyter session works smoothly.

# Jupyter notebook import

Use the 'Insert | Insert Cell Below' menu item to create a new cell.  
Enter just this code and evaluate the cell.

```
1 import my_jupyter
```

This is the first-time import of the file you just created. Note that this will only work if both the notebook and the file are in the same directory (which should be the lab-00 directory).

## Jupyter notebook plotting

Use the 'Insert | Insert Cell Below' menu item to create another new cell. Enter this code and evaluate the cell.

```
1 importlib.reload(my_jupyter) # Reload the module
2
3 d = my_jupyter.gen_data()    # Call the function
4
5 plt.figure(figsize=(18, 6), facecolor='w', edgecolor='w')
6 plt.plot(d[0], d[1])
```

This will plot the data; the `gen_data()` function returned a tuple (x, y), so `d[0]` is x and `d[1]` is y. Thus the `plot` function works quite like the Matlab `plot` function.

# Jupyter notebook iteration

Iterating on data generation:

- In the Codiad / editor tab, edit the function `gen_data` and save
- Go back to the Jupyter tab
- Re-evaluate the third cell to plot the new data

Experiment with changing the values in this command to change the plot appearance:

```
1 plt.figure(figsize=(18, 6), facecolor='w', edgecolor='w')
```



# Jupyter assessment

It is possible, and very common, to write function definitions directly in the Jupyter notebook, instead of in a separate editor.

However, for this course you will be required to write functions in an separate file, outside the notebook, as we want to assess your work as if it was a standalone module.

## Wrapping up

- Go back to the terminal tab
- Enter `git status` to check the status
- Enter `git add lab-00/my_jupyter.py` to stage your new file for addition to your git repo.
- Also add the notebook file.
- Do `git commit -m "Jupyter plotting."` to commit the new files.
- Do `git push` to save your work to the central repo.

Note that your local code will be lost when your cluster is shut down!

# The End