Introduction to Python

GEOFF FRENCH

UNIVERSITY OF EAST ANGLIA

What we'll cover

Python

PyCharm

Getting Python to say 'hello'

Using Python interactively

Using Python as a calculator

Working with text (strings)

Tuples, lists and dictionaries

Conditionals

Looping

Visualising how Python executes code

Functions

List comprehensions

Date and time

NumPy

Matplotlib

Pandas

Python

Python

Python is an interpreted programming language

Programs are stored as plain text files

Python executes your programs

- Line by line

Python versions

We are using Python 3

Python versions

Python 2.x was dominant for a very long time

There were problems with the language

Python 3 fixes many of them

- Released December 2008

Gotcha:

- Some incompatibilities between Python 2.x and Python 3.x
- Why? … Necessary to fix problems in the language

Python 3.x is the future.

PyCharm

PyCharm

PyCharm is an integrated development environment (IDE) from Jetbrains

Designed for Python

There are free community and educational editions you can download (Google ‘PyCharm’)

It highlights syntax errors in your code; basically helps you a lot

You can run it on a Raspberry Pi 3

- Should be pre-installed on the images that we give you
- At home: download the Linux version

Getting Python to say ‘hello’

Getting Python to say ‘hello’

Exercise:

- Write a simple Python program in PyCharm that says ‘Hello world’
- Run it in PyCharm
- Run it from a ‘command prompt’ window
  - You will be running programs from the command line on a Raspberry Pi
    - Not feasible to run programs from within the IDE all the time

Getting Python to say ‘hello’

Exercise:

- Write a simple Python program in PyCharm that says ‘Hello world’
- Run it in PyCharm
- Run it from a ‘command prompt’ window
  - You will be running programs from the command line on a Raspberry Pi
    - Not feasible to run programs from within the IDE all the time

Getting Python to say ‘hello’

Start PyCharm and create a new project

Create a new project

- File menu ▷ New Project…

Choose ‘Pure Python’ on the left

The ‘Location’ (main panel) determines where it will be stored and the last part after the ‘/’ or ‘\’ will determine the name; change it to ‘C:\temp\hello_world’

Choose a Python 3 interpreter; next to ‘Interpreter’ ensure that the selected option contains ‘3.*x*’.

Click ‘Create’

Getting Python to say ‘hello’

Create a python program file called ‘hello_world.py’

In the pane on the left, you should see some tabs down the left hand side; ‘Project’ and ‘Structure’. Ensure ‘Project’ is chosen.

In the large mostly-blank area in the left pane you should see a folder icon named ‘hello_world’.

Right-click ▷ New ▷ Python File

In the ‘New Python file’ dialog that appears:

- Set ‘Name’ to ‘hello_world’
- Click OK

The dialog will disappear and a large blank area will appear in the main pane of PyCharm. A tab at the top will say ‘hello_world.py’

Getting Python to say ‘hello’

Write the program

Enter the text in the main pane:

```
print(‘Hello, world!’)
```

Note: `print` should be lower-case, remember the parentheses and the single quotes will be simple vertical ticks as you are not using MS Office!

Getting Python to say ‘hello’

Run the program

Right-click in the main pane ▷ Run ‘hello_world’ (somewhere in the middle)

A pane will appear and take up the bottom part of PyCharm.

You will see something like:

```
C:\Python25\python.exe C:\temp\hello_world\hello_world.py
Hello, world!

Process finished with exit code 0
```

Getting Python to say ‘hello’

Run the program from the command prompt

Start command prompt:

- Press windows key ▷ type ‘cmd’ ▷ press enter

In the PyCharm ‘Run’ pane, select the text directory containing your program (‘C:\temp\hello_world\’) and press ‘CTRL + C’ to copy.

In the command prompt, type ‘cd ‘ (remember the space) then right click ▷ Paste.

Press enter.

The prompt in the command prompt should change indicating that you have changed directory.

Getting Python to say ‘hello’

Run the program from the command prompt (contd.)

Type ‘python hello_world.py’.

You should see:

```
C:\temp\hello_world>python hello_world.py
Hello, world!
C:\temp\hello_world>
```

That’s it!

Getting Python to say ‘hello’

Extend it a little…

Add another line of code below the ”print(‘Hello, world!’)” line:

```
print(‘My name is <insert name here>’)
```

Using Python interactively

Using Python interactively

Programmers make lots of mistakes

Programs almost never work first time

Lots of time is spent understanding what doesn’t work and how to fix it

Using Python interactively

Normal programming workflow

- Write program
- Run complete program from beginning
- Watch it fail
- Edit program
- Run complete program from beginning … repeat ...

This can be time consuming.

Using Python interactively

Interactive programming

Interactive programming environments allow you to run a single line of code at once in order to test your program in a piecemeal fashion

Much faster turn-around

Using Python interactively

Interactive python from the command prompt

From with the command prompt:

```
C:\temp\hello_world>
```

Type ‘python’:

```
C:\temp\hello_world>
```

Python will start in interactive mode; you can enter code and see the result immediately; type “print(‘Hello, world’)” then press enter.

Using Python interactively

Interactive Python within PyCharm

Along the bottom of PyCharm, look for the small wide buttons ‘Python console’, ‘Terminal’, ‘TODO’, etc.

Click ‘Python console’. The bottom pane will now contain an interactive Python shell like the one in the command prompt.

Using python as a calculator

Using Python as a calculator

From within a Python console (inside PyCharm, or command prompt if you prefer):

Enter ‘5+7’:

```
>>> 5+7
12
```

All the usual arithmetic operations work as you would expect

Using Python as a calculator

Arithmetic operations in Python

Operation	Python	Example
$a + b$	a+b	2+5 = 7
ab	a*b	2*5 = 10
Integer division	a//b	5//2 = 2
$a - b$	a-b	5-2 = 3
$\frac{a}{b}$	a/b	5/2 = 2.5
a^b	a**b	5**2 = 25
$-a$	-a	-(5) = -5

Using Python as a calculator

Assigning values to variables

Compute speed; assign the distance moved to the variable ‘d’:

```
>>> d = 50.0
```

Assign time taken to variable ‘t’:

```
>>> t = 10.0
```

Compute speed by dividing distance by time:

```
>>> d / t
5.0
```

Using Python as a calculator

More complex maths

Import the ‘math’ module:

```
>>> import math
```

Square root:

```
>>> math.sqrt(4.0)
2.0
```

Compute area of circle of radius 10:

```
>>> 10.0**2 * math.pi
314.159265359
```

Working with text (strings)

Working with text (strings)

Text in quotes – either single or double quotes – makes a string:

```
>>> 'Hello world'
'Hello world'
>>> "Hello world"
'Hello world'
```

Adding strings joins them:

```
>>> 'Hello ' + 'world'
'Hello world'
```

Working with text (strings)

Multiplying strings duplicates them:

```
>>> 'Hello ' * 5
```

'Hello Hello Hello Hello Hello'

Working with text (strings)

Composing text

How do we include the contents/value of a variable in a string?

```
>>> radius = 5.2
>>> area = radius**2 * math.pi
>>> print('The area of a circle of radius {} is {:.2f}'.format(radius, area))
The area of a circle of radius 5.2 is 84.95
```

Working with text (strings)

Composing text

```
>>> print('The area of a circle of radius {} is {:.2f}'.format(radius, area))
The area of a circle of radius 5.2 is 84.95
```

The '{}' in the string is replaced by the value of the first argument to 'format(first_arg, second_arg)', hence 5.2 appears in the text.

Working with text (strings)

Composing text

```
>>> print('The area of a circle of radius {} is {:.2f}'.format(radius, area))
The area of a circle of radius 5.2 is 84.95
```

The '{:.2f}' in the string contains formatting information; '.2' states that real numbers should be rounded to 2 decimal places and the 'f' states that it is a floating-point (real) number. It is replaced by the value of the second argument to 'format(first_arg, second_arg)', hence 84.59 appears in the text.

Simple values

Simple values

So far we have seen:

- Integers (whole numbers) e.g. 5
- Real numbers e.g. 3.14
  - Note that these are technically 'floating point' numbers. In practice, a floating point number has limited precision e.g. a python `float` can store around 16 significant figures of precision; after that any digits will be rounded.
- Strings (text)

You can also have

- Booleans
  - Represent a 'truth' value; `True` or `False` e.g. 'yes' or 'no'

Tuples, lists and dictionaries

Tuples, lists and dictionaries

The simple values we have seen so far are useful but limited; just a single value.

What about collections of values?

- How about pairs or triples of values where is makes sense to gather them as one?
  - Beer brews best when the temperature is between a lower and upper bound of 16 and 18 degrees
- What about building a sequence of values as we collect them?
  - We measure the temperature of the beer every minute and over 1 hour; our values are 35, 36, 37, 36, 35, 38, 37, 36, ….
- What about associating values with one another?
  - The best temperatures for brewing beer given these strains of yeast are:
  - It would be nice to be able to retrieve the value for a given strain…
- How about checking to see if we are aware of some value?
  - These are the yeast strains we know about…

Strain	Best temp (C)
Wyeast	21
Lallemand	17
Siebel inst.	12

Tuples, lists and dictionaries

Tuples

For pairs, triples, … n-tuples of values, use Python tuples; surround them in parentheses and separate values with commas:

```
temp_range = (16.0, 18.0)
```

Values don't have to be of the same type:

```
successful_experiment = ('success', 16.2, 13)
```

Tuples can be nested:

```
successful_experiment = ('success', ('Wyeast', 18.2))
```

Tuples, lists and dictionaries

Tuples

```
successful_experiment = ('success', 16.2, 13)
```

To get an element at a specific position:

```
print(successful_experiment[0])
```

```
success
```

```
print(successful_experiment[1])
```

```
16.2
```

Tuples, lists and dictionaries

Tuples

Tuples cannot be modified once they are created; the elements cannot be changed and their length cannot be extended.

Tuples, lists and dictionaries

Tuples

Operation	Example
Get an element at a specific position	`xs[1]`
Get a range of elements	`xs[1:3]`
Get the length of the tuple	`len(xs)`
Check for the presence of an element	`'hi' in xs`
Check for the absence of an element	`'hi' not in xs`

Tuples, lists and dictionaries

Lists

Python lists are like tuples that can be modified. Notice the use of square brackets when creating them:

```
temp_measurements = [16.0, 17.2, 18.2]

temp_measurements[1] = 17.6

print(temp_measurements)
```

```
[16.0, 17.6, 18.2]
```

Note that indices/positions are 0-based.

Tuples, lists and dictionaries

Lists

The same operations – `xs[2]`, `len(xs)`, `'hi' in xs`, `'hi' not in xs` – work with lists as they do with tuples.

Where tuples cannot be modified once created, lists can. They can also grow and shrink as needed; you can add and remove elements.

Tuples, lists and dictionaries

Lists

Operation	Example
Append an element	l.append(20)
Append all elements from another list	l.extend([10, 20, 30])
Insert element at specific position	l.insert(2, 'hi')
Remove all elements	l.clear()
Remove a specific element by value	l.remove('hi')
Set an element	l[2] = 'there'
Delete an element	del l[2]

Tuples, lists and dictionaries

Dictionaries

Dictionaries are simple tables that allow you to look things up.

Each entry in a dictionary associates a *key* with a *value*.

In this case – considering our beer example – we are mapping strings/text – for yeast strains - to real numbers - temperatures. You don't have to use strings and numbers, you can use anything you like as the key or value.

There is one limitation; keys must be *immutable*; their value should not be able to change. So strings, numbers, booleans, tuples are examples of objects that can be used as keys, but lists, sets and dictionaries cannot be used as keys.

Tuples, lists and dictionaries

Dictionaries

Associate key to value with a colon, separate key-value entries from one another with commas, all inside curly braces:

```
strain_to_temp = {'Wyeast': 21, 'Lallemand': 17, 'Siebel inst.': 12, 'non-existent': 1200}
```

To get the value associated with a key, use the get operation (like with lists):

```
print(strain_to_temp ['Lallemand'])
```

```
17
```

Key	Value
Wyeast	21
Lallemand	17
Siebel inst.	12
Non-existent	1200

Tuples, lists and dictionaries

Dictionaries

Associate key to value with a colon, separate key-value entries from one another with commas, all inside curly braces:

```
strain_to_temp = {'Wyeast': 21, 'Lallemand': 17, 'Siebel inst.': 120, 'non-existent': 1200}
```

Changing the value associated with a key is done using the set operation, like with lists:

```
strain_to_temp['Siebel inst.'] = 12
```

You can delete entries with 'del', like with lists:

```
del strain_to_temp['non-existent']
```

Key	Value
Wyeast	21
Lallemand	17
Siebel inst.	12

Tuples, lists and dictionaries

Dictionaries

Operation	Example
Get a value associated with a key	table['hi']
Get the length of the tuple or list	len(table)
Check for the presence of an element	'hi' in table
Check for the absence of an element	'hi' not in table

Conditionals

Conditionals

If statements

So far, we've seen how to write simple programs that perform some actions in linear order.

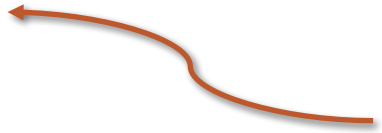
Lets use an if-statement to implement this decision table:

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Conditionals

If statements

```
t = get_temperature_reading()
if t < 10.0:
    turn_on_heater()
```



NOTE THE INDENTATION!!!!

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Conditionals

If statements

```
import math, time
t = 12.0 + math.sin(time.time()) * 5.0
if t < 10.0:
    print('Turn heater on')
```

Now what?

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Conditionals

If statements

```
import math, time
t = 12.0 + math.sin(time.time()) * 5.0
if t < 10.0:
    print('Turn heater on')
else:
    # This will be execute if t >= 10.0
    ...
```

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Conditionals

If statements

```
import math, time
t = 12.0 + math.sin(time.time()) * 5.0
if t < 10.0:
    print('Turn heater on')
else:
    if t > 14.0:
        print('Turn heater off')
    else:
        pass
```

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Conditionals

If statements

```
import math, time
t = 12.0 + math.sin(time.time()) * 5.0
if t < 10.0:
    print('Turn heater on')
elif t > 14.0:
    print('Turn heater off')
else:
    pass
```

Temperature reading	Action
< 10	Turn on heater
> 14	Turn off heater
	Do nothing

Indentation

A quick note about indentation:

Python's blocks – if statements, while loops, etc – are delimited by changes in indentation.

The typical Python style is 4 spaces per indent level. PyCharm does this by default.

```
if t < 10.0:
    print('Turn heater on')
elif t > 14.0:
    print('Turn heater off')
print('...')
```

Comparisons and tests

Here are Python's comparison and test operators:

Comparison	Python
Equal	a == b
Not equal	a != b
Less than	a < b
Less than or equal	a <= b
Greater than	a > b
Greater than or equal	a >= b
Identity (same object)	a is b
Inverted identity	a is not b

Comparison	Example
Containment	a in b
Inverted cont.	a not in b

Test	Example
Not (invert)	not state
And	a == b and c == d
Or	a == b or c == d

Looping

Looping

While loops

While loops run the code within them *while* a condition remains to be satisfied. When it is no longer satisfied, the loop ends.

```
while still_brewing():
    t = 12.0 + math.sin(time.time()) * 5.0
    if t > .....
```

Looping

For loops

Recall the collections discussed earlier; tuples, lists and dictionaries.

How about taking each element in turn and performing some action?

Looping

For loops

Note that readings is a list of tuples; each tuple is a (time, temperature) pair.

```
for reading_time, reading_temp in readings:
    print('At time {} the tempreature was {}'.format(reading_time, reading_temp))
```

The for-loop will take each elements from the collection in turn, process it, and finish when there are no elements left.

Visualising how Python executes code

Visualising how Python executes code

To see how Python – and other programming languages – execute code, check out:

http://www.pythontutor.com/visualize.html

Functions

Functions

In the lab examples you will have seen function calls like:

```
GPIO.output(22, False)
```

Functions improve readability and maintainability of you code by:

- hiding complex operations in a relatively simple looking piece of code like the one above
  - while the Pi's GPIO ports are quite simple, capturing an image from a camera – for example – is somewhat more involved
- reducing repetition; write your code once and use it many times throughout your code

Functions

Example function

```
import math, time


def get_temperature_reading():
    t = 12.0 + math.sin(time.time()) * 5.0
    return t
... further down ...
temperature = get_temperature_reading()
...
```

Functions

Parameters

You can pass values (parameters) to a function to control its operation:

```
def update_heater(cur_temp, min_temp, max_temp):
    if cur_temp < min_temp:
        turn_heater_on()
    elif cur_temp > max_temp:
        turn_heater_off()
```

Functions

Parameters

Pass the parameters in the call:

```
t = get_temperature_reading()
temp_range = yeast_temperatures[yeast_name]
update_heater(t, temp_range[0], temp_range[1])
```

List comprehensions

List comprehensions

List comprehensions are a bit of ‘syntactic sugar’ to simplify operations involving processing lists in Python.

You can apply transformations:

In mathematical notation:

$y = 1 \cdots 100$

$z = [2x : x \in y\ ]$

In Python:

```
y = range(1, 101)
z = [x*2  for x in y]
```

List comprehensions

List comprehensions are a bit of ‘syntactic sugar’ to simplify operations involving processing lists in Python.

You can filter by a condition:

In mathematical notation:

$y = 1 \cdots 100$

$z = [x : x \in y, x > 2, x < 9]$

In Python:

```
y = range(1, 101)
z = [x  for x in y  if x > 2 and x < 9]
```

List comprehensions

List comprehensions are a bit of ‘syntactic sugar’ to simplify operations involving processing lists in Python.

A combination

In mathematical notation:

$y = 1 \cdots 100$

$z = [x^2 : x \in y, x > 2, x < 9]$

In Python:

```
y = range(1, 101)
z = [x**2  for x in y  if x > 2 and x < 9]
```

Date and time

Date and time

Acquire the date and time using the datetime module.

```
import datetime

right_now = datetime.datetime.now()

todays_date = datetime.date.today()
```

NumPy

NumPy

NumPy adds Matlab style array/matrix functionality to Python.

Makes handling large arrays of numeric data fast – in terms of performance – and convenient – in terms of code.

```
import numpy
```

Or:

```
import numpy as  np
```

NumPy

NumPy arrays are like Matlab arrays:

- once created, you cannot change the size of the array; to grow it you create a larger array and copy elements over
- can have up to 32 dimensions

In contrast to Python lists:

- much faster than lists for handling large quantities of numeric data
- lots of convenient functions for doing numerical work typical in maths and science

NumPy

Convert a list to a NumPy array:

```
xs = [1, 2, 3, 4, 5]

x = np.array(xs)
```

Or:

```
x = np.array([1, 2, 3, 4, 5])
```

NumPy

Retrieving elements:

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])

first = x[0]

second = x[1]

last = x[-1]
```

Retrieving a range of elements:

```
two_to_four = x[1:4]

inner = x[1:-1]
```

NumPy

You can create a 2D array like so:

```
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

You need two indices to access elements:

```
sub_array = x[0:2, 1:3]
```

To have all the elements in one of the dimensions:

```
part_of_first = x[0:2, :]
```

NumPy

Arithmetic operators work on NumPy arrays:

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([9, 8, 7, 6, 5])

Z = x + y*2 - 3
```

Like Matlab, NumPy will ‘broadcast’ the scalar numbers and perform them over the complete array.

NumPy

GOTCHA for Matlab users; all arithmetic operators work in an element-wise fashion, so:

```
x = np.array([1, 2, 3, 4, 5])

y = np.array([9, 8, 7, 6, 5])

x * y
```

Will results in the array [9, 16, 21, 24, 25]; not matrix multiplication. To multiply matrices, call NumPy's dot function:

```
z = np.dot(x, y)
```

NumPy

There is far too much to NumPy to cover in a single lecture, so we can only introduce it here.

Full documentation is available here:

http://docs.scipy.org/doc/

Also, Google and Stack Overflow are VERY helpful…

Matplotlib

Matplotlib

A very quick intro, just a basic plot:

```
import numpy as np
from matplotlib import pyplot as plt
x = np.arange(-25.0, 25.0, 0.05)
y = np.sin(x) * x**2
plt.plot(x, y)
plt.show()
```

Matplotlib

Full documentation is available here:

http://matplotlib.org/

Pandas

Pandas

Another huge package with lots of features, so we're only going to scratch the surface ☺

Good for working with data as tables.

Good export and import facilities; CSV for Excel, HTML, others

Really good tools for selecting subsets of your data and manipulating it.

Pandas dataframes are very similar to data frames in R…

Pandas

```
import datetime
import numpy as np
import pandas as pd


# Time offsets; half-second interval up to 100 seconds
dt = np.arange(0, 100.0, 0.5)
# Time stamps; we're just going to fake some for now
now = datetime.datetime.now()
ts = [now + datetime.timedelta(seconds=x) for x in dt]
# Temperature readings
temp = 12.0 + np.sin(dt) * 5.0
```

Pandas

The zip function takes a number of sequences (tuples or lists) and constructs tuples out of corresponding elements, so the first tuple of zip(ts, temp) will be (ts[0], temp[0]), then (ts[1], temp[1])...

As a consequence we can use to to pair out timestamps and temperatures to make rows:

```
rows = zip(ts, temp)
```

Pandas

Now, lets make a data frame (note that DataFrame requires that the data is in the form of a list, so convert it first as zip returns an iterator):

```
rows = list(rows)

df = pd.DataFrame(columns=['Timestamp', 'Temperature'], data=rows)
```

Pandas

You can export a dataframe to a CSV file for loading in Excel:

```
df.to_csv('my_data.csv')
```

Will write the table to the file 'my_data.csv'.

Pandas

Selecting subsets

Select columns by name:

```
temperatures = df['Temperature']
```

Now to select only rows where the temperature is > 16:

```
temp_over_16 = df[df['Temperature'] > 16]
```

Thank you!