Introduction to Python

What is Python

The Python programming language is an interpreted language ideally suited to scripting small utility program. It was created in the early 1990s by Guido van Rossum and is named after the BBC comedy series "Monty Python's Flying Circus"; it has no connection to large snakes. The language is often compared to `Perl` (Pattern Extraction and Reporting Language) which is also widely used for scripting, but it differs in a number of important ways. Perl provides a multitude of different way to perform tasks. While this shows flexibility it can result in code which is difficult to maintain, and through the use of regular expressions can be almost cryptic in it's method of operation. Extending the language has also proved to be complicated in Perl where the addition of object orientated extensions, widely used in the bioinformatic libraries, often has a *ad hoc* appearance. Python was designed as an object orientated language from the ground up, with a strong emphasis on readable and easily documented code. The BioPython libraries provide similar functionality to that of BioPerl for bioinformatics applications.

The Interactive Prompt

Python can be run interactively at the command prompt or more commonly as a script file. For an example of interactive Python, at the Unix command prompt start the interpreter (note [enter] means press the return or enter key):

```
$ python [enter]
```

You will get the prompt, after a few lines of version text:

```
>>>
```

Type

```
>>> print "Hello World" [enter]
```

The interpreter will respond:

```
Hello World
```

Another example, type the next line - with the enter at the end as before:

```
>>> print 2 ** 8
256
```

You can type as many commands as you like and each is ran immediately after it is entered. To exit the interactive session type Control D (Control key and D at the same time).

As a Script File

To run Python from a script involves a couple of more steps but it's the most convenient way to perform real world tasks. A script can be run in two ways.

In the first way, the script is run by typing `python` at the command prompt and then the name of the script file. For example:

```
$ python myscript.py
```

Note that the Python script file has the suffix `.py`. This command will execute the python interpreter and then your script.

Another method of running Python scripts is to make the first line of your script the path to the python interpreter. On the MSE UNIX servers, the path is `/usr/bin/python`. To find out what the path is on any UNIX machine, type the command `whereis python` at the command prompt and the path will print out on the screen. The Python path is preceded with the characters `#!`, as in the example below.

```
#!/usr/bin/python
#
# This a comment - maybe a description of what the script does
#
#
print 'Always look on the bright side of life'
```

This simple script only does one thing - print a line of text. Write and save the above six lines to a file and call it `brian.py`. To execute the file on a UNIX machine we first need to change its permissions to executable. This is done by running the command:

```
$ chmod +x brian.py
```

This changes the permission of the file "brian.py" to executable.

The script can now be ran at the command prompt by typing:

```
$ ./brian.py
```

This should result in the line "Always look on the bright side of life" printed on the screen. Note that the characters "./" in front of the script name tell the operating system that the script file is located in the directory you are currently in.

Python Core Data Types

- *Numbers* can be integers such as 567, floating point 3.143 or even complex $3 + 4j$. Number operators are the normal $+ - */$. For exponentation use `**`. If at the top of a script you include the line `import math` you can then use functions such as `sqrt`, `log` and access `pi` in the `math` library:

  ```
  #!/usr/bin/python
  ```

```
import math
x = 456.657
y = math.sqrt(x)

# z is equal to the log of 5.7687 to the base 10
z = math.log(5.7687,10)

circ = 2*math.pi*x

print circ
```

- *Strings* Textual information is stored in strings, an ordered collection of characters. Any line of text characters is a string , for example "Hello World", "This is a description of a program" or "AGCTTAGCTAGCTGT". Strings can be manipulated in many ways. It may be something simple like changing the first letter of each word to upper case, or for a string of DNA symbols, calculating the proportion of G plus C nucleotides. Examples of accessing single characters,

```
#!/usr/bin/python

# assign a nucleotide string to the variable S
S = 'ACGTACGTACGT'
x = len(S)
# the variable x holds the length of string S, in this case x = 12
# use square brackets [] to access individual character in the string
char1 = S[0]
# char1 is "A". Note that the first character in the string
# is always at index 0. To get the last character
char12 = S[11]
# or a shortcut
char12 = S[-1]
# The third character in the string
char3 = S[2]
# The second last character in the string
char12 = S[-2]
```

Sections of strings or substrings may be extracted by the use of slices. A slice can have two numbers within the square brackets. Their general form `X[I:J]` means extract everything from string `X` from offset `I` up to but *not including* offset `J`.

```
#!/usr/bin/python

# declare a string
S = 'Spam'
# to extract the first 3 characters of S, characters at positions
# 0 to 2
x1 = S[0:3]
# gives us "Spa". A shortcut for the same result
```

```
x1 = S[:3]
# for the other end of "Spam". This gives us everything but the first
# letter ("pam")
x2 = S[1:]
# To copy all of string S1 to another string S2 use
S2 = S1[:]
# note that this is different to S2 = S1. S2 = S1[:] creates
# a whole new string with the same characters as S1. S2 = S1
# means that the variable S2 points to string S1. A new string is not
# created.
```

To join two strings together (concatenate), use the + symbol

```
#!/usr/bin/python

# declare a string
S = "Spam"
S2 = S + 'xyz'
# S2 now equals "Spamxyz"
# to repeat a string, use the * symbol
S3 = S*2
# S3 is equal to "SpamSpam"
```

Note that Python strings are immutable. This means that once a string is created it cannot be changed, only copied. For example:

```
#!/usr/bin/python

# declare a string
S = 'Spam'
S[0] = 'Z'
# This will give an error, telling you that  you could not make
# the assignment  - you are trying to change string S.
# The solution is to make a new string
S2 = 'Z' + S[1:]
# S2 is equal to "Z" plus all of the characters from position
# 1 to the end of the string, leaving out position 0.
```

Often you need to find the location of a small string, a substring, within a larger string. This often used to extract information from lines of text. For example:

```
#!/usr/bin/python

# declare a string
S = "The search for the holy grail"
index = S.find('for')
print index
# index will be equal to 11 as the substring 'for'
# starts at the 11th characters from the beginning of S
# If the substring is not in the string, the function returns -1
# For example
index = S.find('spam')
# index will hold the value -1 as the string 'spam' does not exist in S
```

The find() function could be used in bioinformatics to find the location of a specific motif in sequence string.

Another common operation on a string is to replace a substring. The replace() function is used to achieve this.

```
#!/usr/bin/python

# declare a string
S = "The search for the holy grail"
# replace a substring
R = S.replace('search','quest)
# R now equals 'The quest for the holy grail'
# Note that when replacing a string you need to create a
# new string, in this case R.
```

Often you will need to work with strings from comma delimited databases, or maybe delimited output from one of the many bioinformatics programs.

```
#!/usr/bin/python

# a comma delimited line
C = '4325, 6754, A sequence description,1e-12'
# The string 'C' could be the output from a Blast comparison with 4
# fields; match begin, match end, query description and comparison e value in
# scientific notation.
F = C.split(',')
x1 = F[0]
x2 = F[1]
x3 = F[2];
print x1, x2, x3
# When the split() function is applied to the string 'C', it splits
# the string at the delimiter (',') and produces a list of substrings (F).
# Each of the substrings can then be accessed and treated as a normal string.
# Any character can be used as a delimiter including tabs ('\t') and newline
# characters ('\n').
```

When lines of text are read from a file, before each line can be processed, the newline character must be removed. The `rstrip` function removes all whitespace character (tab, newline, space etc) from the *end* of the line.

```
#!/usr/bin/python

# A string like a line read from a file
line = '4325, 6754, A sequence description,1e-12\n'
line = line.rstrip()
print line
# In this example the newline at the end of the 'line' string
# is removed and the new string is assigned to a new line variable.
```

- *Lists* are ordered collections of objects, like arrays in other languages. They can contain a mixture of types, for example number and strings. Lists are mutable, unlike strings, so elements in the list can be changed without making a new copy. Some examples:

```
#!/usr/bin/python

# Declare a list
L = [123,'Spam',3.142]
# a holds the length of the list (3)
a = len(L)
n # b holds the first list element (123)
b = L[0]
# c holds all of L except the last element
c = L[:-1]
# d holds all of L except the first element
d = L[1:];
# e holds the concatenation of list L and [4,5,8]
e = L + [4,5,8]
# f holds the length of the joined lists (6)
f = len(e)
print a,b,c,d,e
```

Because lists are mutable you can add to them.

```
#!/usr/bin/python

# Declare a list
L = [123,'Spam',3.142]
L.append('ABC')
print L
# L is now equal to [123,'Spam',3.142,'ABC']
```

Lists have bounds testing so that if you attempt to access a list element which doesn't exist, you will get an error. For example in the above list L, trying to access `L[21]` will result in an error.

- *Dictionaries* are one of the most useful of all data structures. Where lists and strings store items in an ordered arrangement, dictionaries use mappings to place items in a data structure using keys.

```
#!/usr/bin/python

# Declare some elements in a dictionary
D = {'food': 'Spam', 'quantity': 4, 'Colour': 'pink'}
# the keys in this mapping are 'food', 'quantity' and 'colour'
a = D['food']
# a holds the data accessed by the key 'food' ('Spam')
# Dictionaries are mutable so you can change elements
D['quantity']+= 1
# the dictionary D is now {'food': 'Spam', 'quantity': 5, 'colour': 'pink'}
# Create a new dictionary
D2 = {}
# add a key value pair
D2['name'] = 'Bob'
```

```
# add another
D2['age'] = 40
# D2 is now {'age': 40, 'name': 'Bob'}

# To check if a key is in a dictionary, you can use an if test

# if D doesn't include the key 'f'
if not D.has_key('f'):
    print 'missing'
```

While dictionaries do not store data in any ordered fashion, the keys can be sorted before retrieving the values.

```
#!/usr/bin/python

# Declare some elements in a dictionary
D = {'a': 1, 'c': 2, 'b': 3}
# get the keys from the dictionary 'D'
Ks = D.keys()
# K now holds all the keys for D, ['a','c','b']
Ks.sort()
# sort the key list, ['a','b','c']

# To print out all of the dictionary using the for loop,
for key in Ks:
    print key, D[key]
```

File handling

The Python file functions make it easy to read and write a file. Here are some examples.

```
#!/usr/bin/python

# create a new file in output mode.
f = open('data.txt','w')
# write the word 'Hello' to the file 'data.txt'
# Note the newline character (\n) at the end of each string
f.write('Hello\n')
f.write('World\n')
f.close()
```

To read data from a file, we use the open function again.

```
#!/usr/bin/python

# open the file for reading
f = open('data.txt', 'r')
```

```
# in fact as 'r' is the default mode for open
# you can also use f = open('data.txt')
flist = f.readlines()
# This function reads all the lines of the file into a
# list of strings
```

Control Structures

`Python` as a programming language is unusual in that it uses indentation to lay out control structures such as loops and `if` tests, whereas most other languages use braces. The enforced use of indentation makes the code more readable, although it can occasionally be cumbersome during programming. Indentation can be either tabs or spaces, with the convention being tabs.

- *if* statement.

  ```
  #!/usr/bin/python

  # use if to compare x and y
  x = 4
  y = 5
  if x == y:
      print "x equals y"
  else:
      print "x does not equal y"
  ```

  To compare text strings use the same syntax.

  ```
  #!/usr/bin/python

  # use if to compare x with different strings
  # note the colon, then the indentation after
  # the if and elif test, and the single colon
  # after the else statement
  x = "rabbits"
  y = "foxes"
  if x == y:
      print "x and y have the same text"
  elif x == "bugs":
      print "x is a bug"
  elif x == "spiders":
      print "x is a spider"
  else:
      print "x is not equal to bugs, spiders or" + y
  ```

- The *for* loop is the main means of accessing multiple items in a list or string.

  ```
  #!/usr/bin/python
  ```

```
L = ["Red","Yellow","Green"]
# L is a list of strings
# print each of the strings in L
for x in L:
    print x
```

A *for* loop can be used to process all the lines in a file. This could be a FASTA format sequence file with a header and multiple lines of nucleotide characters.

```
#!/usr/bin/python

# open a FASTA sequence file
f = open('seq01.fasta')
# read all the lines into a list of strings
flist = f.readlines()

# for each line in the list
for line in flist:
    # remove the newline character at the end of each line
    line = line.rstrip()
    # print the line
    print line
```

We have already used a *for* loop to access values in a dictionary; here is another example.

```
#!/usr/bin/python

# Here is dictionary where the key is a floating point number
# and the value a file name
D = {1.456: "file01.txt", 0.845: "file09.txt",
    2.675: "file06.txt", 1.345: "file02.txt" }
# get the keys from the dictionary 'D'
K = D.keys()
# K now holds all the keys for D
Ks.sort()
# sort the key list

# print out all of the dictionary using the for loop,
# numerically sort by key
for key in Ks:
    print key, D[key]
```

- The *while* loop repeatedly executes a block of indented statements as long as the test at the top is true.

```
#!/usr/bin/python
```

```
a = 0
b = 10
    # keep printing a while a is less than b
    while a < b:
        # the comma at the end of the line means
        # print a without a newline character
        print a,
        # same as a = a + 1
        a += 1
    # output is "0 1 2 3 4 5 6 7 8 9"
```

This next example shows how a *while* loop can be used read lines of data from the command line or from a redirected file. Each line is printed with a line number.

```
#!/usr/bin/python

# get a line from the commandline (stdin)
line = raw_input()
# set the counter to 1
num = 1
# while "line" is not an empty string
while line:
    # strip the newline character from the end of line
    line = line.rstrip()
    # print the line along with its line number
    print "Line ",
    print num,
    print " " + line
    # get the next line
    line = raw_input()
    # increment the line counter
    num += 1
```

Exceptions

Exceptions are used in `Python` to handle error conditions. In effect it allows you to jump out of a large chunk of code to handle an error which has occurred. If it is a serious error, we may need to end the program, or maybe an error message needs to shown to the user. Exception handling is common in modern computer languages, and can used in Python to handle errors gracefully, allowing scripts to exit gracefully when there is an error, informing the user of the reason for the exit.

So, what sort of errors can raise an exception? The most common that you will see, are in reference to file handling. If a script tries to open a file and it doesn't exist, an exception will be raised. An exception will also be raised when there are errors when checking for type conversion. For example reading a file of numbers; if there is text amongst the numbers, the script will try and perform

arithmetic on words instead on numbers and an exception will be raised. If the exception is not caught, the program will fail. Exception handling allows us to catch the error state, either correct it, or end the program gracefully. An example:

```
#!/usr/bin/python

# continous loop to read characters from the command line
# Note: this example can be improved, see text and the following example
while True:

    try:

        # raise an exception if there is an error
        line = raw_input()

        line = line.upper()

    # get out of the while loop if raw_input() gives an error or
    # end of the file is reached (Control-D from screen)
    except:
        break

    print line
# end while
```

We place the statements that may raise an exception, in this case `line = raw_input()` between `try` and `except` statements. This means that any error in this region will raise an exception and the script will instantly jump to `except` and execute the statement following it.

Another example:

```
#!/usr/bin/python

# continous loop to read characters from the command line
while True:

    reply = line = raw_input('Enter text: ')
    if reply == 'stop':
        break
    # end of if

    try:

        num = int(reply)
    except:
        print "You didn't enter a number"

    square = num ** 2
    print square

# end while
```

The example above reads a number from the command line, squares it, then prints it out. Note that the number read from the command line is a string, which has to be converted to an integer number. Hence the "num = int(reply)". If "reply" is a number, the script squares it and prints it out. But if you enter "hello", it cannot convert it to a number, so it raises and exception. When the exception is raised, the code jumps straight to the `except` statement where it prints an error.

There is still a problem with this example though. If you don't type a number, while it does not try to square the non-number, it keeps printing the result of the previous successful number squaring. We need a way to say print the number squared only if there isn't an exception. The next example uses an `else` statement to get around this problem.

```
#!/usr/bin/python

# continous loop to read characters from the command line
while True:

    reply = line = raw_input('Enter text: ')
    if reply == 'stop':
        break


    try:

        num = int(reply)
    except:
        print "You didn't enter a number"

    else:
        square = num ** 2
        print square
    # end of if-else


# end while
```

This time when we convert the "reply" string to an integer number, if an exception occurs (the string is not a number), we print an error message. If the "reply" string is a number (the "else:" part) square it and print it out.

In assignments this semester the only time you will be expected to use `try, except` statements is in simple file handling. See the examples above or the python example scripts.

Links to further information

`http://www.python.org/doc/` version 2.x
`http://docs.python.org/2/tutorial/`
`http://www.sthurlow.com/python/`

There is also an easy-start tutorial from wikibooks, that can be found on the →Lab Documents.

R. Hall, Feb 2013
revised, L. Stern Feb 2015