Programming with Python

Dr Derek Huntley

Imperial College Bioinformatics Centre

Imperial College London

2017

d.huntley@imperial.ac.uk

Which Language Do I Use?

Each language has it's own niche and practical features.

File manipulation, text handling – Perl

More complex data manipulation and mathematical modelling – Python

More robust and complex data manipulation – C/C++

Graphical requirement – Used to be Java but now HTML/JavaScript

These are very vague guidelines and all of the languages overlap in their abilities.

Choose the language that suits your needs.

Programming

Programming is just a collection of procedures to solve a particular problem

The challenge with programming is not the code itself, the syntax, but the algorithm it is implementing

Once you can write one programming language you can adapt to almost any other language easily

The first step in designing a program is to formulate the algorithm

This can most easily be achieved with pseudocode

Python

Python should be available on any Unix/Linux machine and it is on these systems that you are encouraged to learn.

A Windows version is freely available from ActiveState (ActivePython) and this includes an IDE.

http://www.activestate.com

Commenting Python

Python code lines can be commented with the #

An exception is the shbang line which also starts with a #

Officially there is no multi-line comment although you can use triple quotes:

```
‘’’
A comment…
Continued comment …
‘’’
```

Technically though this is a docstring and not encouraged

Any code line that starts with # will be ignored by the interpreter

It is good practice to comment code. This not only helps other people who may have to use your code it also helps when you have to back and look at your own!

Writing Python

The simplest command in any programming language is to print something to screen (standard output).

```
# A comment line …

print ("Hello, World!“)
```

NOTE: Most other languages, including Perl and Java always end their lines with a semi colon, Python does not. In Perl the line above would be:

```
print "Hello, World!“;
```

Running the Program

The Python script can be created in any test editor.

Save the file with a suitable name and include, by convention, the suffix .py.

To run the script hello.py:

```
python hello.py
```

Output:

```
Hello, World!
```

Data Types

The main purpose of a computer program is to manipulate data. As with all programs data are assigned to named variables in Python, which can then be manipulated.

Compiled languages such as C and Java require you to declare the type of the variable before you assign a value to it. With scripting languages this is unnecessary.

Variables can have any name but it is advisable to make the names meaningful as this can make the code easier to read and understand.

Assigning values to variables is simple:

```
x = 1
name = “Doris”
number = 24
```

Operators

Data, such as numbers, can be manipulated with operators, pretty much as you would expect.

```
number1 = 2
number2 = 34

number3 = number1 + number2

print (number3)
```

This will, not surprisingly, print 36.

Other Mathematical Operators

Addition	1 + 2
Subtraction	4 - 3
Exponentiation	5 ** 2
Multiplication	2 * 3
Division	14 / 3
Remainder	14 % 3
Increment	number+=1
Decrement	number-=1

These operators are the same in other languages.

Augmented Assignment

+=	a += 5	a = a + 5
	str += "bar“	str = str + "bar“
-=a	-= 5	a = a – 5
*=	a *= 5	a = a * 5
	str *= 3	str = str * 3
/=	a /= 5	a = a / 5
**=	a **= 5	a = a ** 5

Compiled Languages

Compiled languages are more rigid in their type declaration and the type has to be specified. In Java the code would be:

```
int number1 = 2;
int number2 = 34;

int number3 = number1 + number2;

System.out.println(“”+number3);
```

In compiled languages you can also not do something like:

```
int number1 = 23;
number1 = “one”;
```

This would be valid in Python or Perl which don’t have strict data typing.

Print Statement

As in most computer languages Python allows an easy way to write to the standard output.

Python's print only accepts output of strings, and if the variable sent to it is not a string it is first converted and then output.

The print always put a line-break ("\n") at the end of the expression to be output.

To prevent this, at the end of the printed string a comma is followed by end=" ":

```
print ("This line does not end with a newline character", end=" ")
```

Print Examples

```
print ("This is a“)
print ("test“)
```

will print

```
This is a
test
```

while

```
print ("This is a", end=" ")
print ("test“)
```

will print

```
This is a test
```

Print Statement

As data types are not identified by notation they cannot be interpolated within quotes.

For example:

```
number1 = 2
number2 = 34

number3 = number1 + number2

print (“Answer is: number3”)
```

Will, not surprisingly, print

```
Answer is: number3
```

Solution

As data types are not identified by notation they cannot be interpolated within quotes, they have to be appended. To do this in Python a comma is used:

```
number1 = 2
number2 = 34

number3 = number1 + number2

print (“Answer is: “, number3)
```

Will now print:

```
Answer is: 36
```

More Examples

```
print ("2 + 2 is", 2 + 2)
print ("3 * 4 is", 3 * 4 )
print ("100 - 1 is", 100 - 1 )
print ("(33 + 2) / 5 + 11.5 is", (33 + 2) / 5 + 11.5 )
```

Output is:

```
2 + 2 is 4
3 * 4 is 12
100 - 1 is 99
(33 + 2) / 5 + 11.5 is 18.5
```

Assigning Values and Printing

```
a = 123.4
b = 'Spam'
first_name = "Bill"
b = 432
c = a + b
```

```
print ("a + b is", c)
print ("first_name is", first_name)
print (“b is ",b)
```

The output:

```
a + b is 555.4
first_name is Bill
b is Spam
```

Comparison Operators

To compare variables:

==	Equal to (NOTE: “=“ is an assignment operator)
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

Flow Control

Flow control is the way instructions are executed by a program.

Most languages have a linear flow control, meaning every line is executed, one at a time, from top to bottom.

This linear flow control can be disrupted by two types of statements: looping and branching.

Looping statements tell the computer to execute a determined set of commands until certain condition is met.

This can be a numeric value (ie from 1 to 100) or the number of items in a list.

Branching statements are also known as conditional statements and tell the computer to execute/or not determined lines depending on certain conditions.

range Command

Ranges are handy for generating a sequence of numbers as a list.

range(10) returns a list of numbers starting from zero counting up to (but not including) ten in steps of one.

range(2, 10) will do the same, only starting from two instead of zero.

range(-3, 3) starts from -3 and ends at 2 (not 3) in steps of one.

The step size can be altered.

```
range(0, 10, 2)  is [0, 2, 4, 6, 8]
range(10, 2, -1)  is [10, 9, 8, 7, 6, 5, 4, 3]
range(-5, 7, 3)  is [-5, -2, 1, 4]
```

The third argument to the range function alters the step size.

Making the step size negative will result in a list that counts down.

For Loops

A ***for loop*** will iterate over a block of code a certain number of defined times. This may be for each item in a list or a set number of times.

The structure of the Python ***for loop*** is very different to other languages.

```
onetoten = range(1,11)
for count in onetoten:
        print (count)
```

Or:

```
for count in range(1,11) :
        print (count)
```

NOTE: This will print out numbers 1 to 10

Defining Control Structure Boundaries

How does the program know the range of the control structure, or code block?

Where does it end?

In other languages, such as Java or Perl, curly braces are used to define the boundaries:

Perl:

```
for($count = 1; $count <= 10; $count++){
        print “$count\n”;
        }
```

Java:

```
for(int count = 1; count <= 10; count++){
        System.out.println(“”+count);
        }
```

Python Version

Python uses indentation to define the boundaries of a code block:

```
onetoten = range(1,11)
for count in onetoten:      # Note colon
            print (count, end = " ")
            print ("Still in code block …" , end = " ")
            print ("Still there …")

print ("Now outside code block")
```

Will print:

```
1 Still in code block … Still there …
2 Still in code block … Still there …
….
10 Still in code block … Still there …
Now outside code block
```

for Loop

Note that the first line of the loop ends in a colon.

This and the word ***for*** in the line tell the interpreter that this a ***for*** loop and the indented block below is the code to be executed repeatedly until the last element in the list is reached.

How Python knows where the loop ends? Indentation. As previously mentioned, many languages use curly braces, parentheses, etc. In Python the loop ends by checking the indentation level of lines.

If this is good or bad is open to interpretation.

Indentation

The use of indentation in python is described as one of its strengths as it produces clear and well defined code that is easy to read.

However, care must be taken and there are issues that must be considered.

Due to of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

Be consistent when using indentation so that code blocks are clearly defined.

Poor Indentation

Here is an example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l):
                # Compute the list of all permutations of l
        if len(l) <= 1:
                                return [l]
        r = [] for i in range(len(l)):
                        s = l[:i] + l[i+1:]
                        p = perm(s)
                        for x in p:
                         r.append(l[i:i+1] + x)
        return r
```

Incorrect Indentation

The following example shows various indentation errors:

```
 def perm(l):                    # error: first line indented
for i in range(len(l)):          # error: not indented
            s = l[:i] + l[i+1:]
                    p = perm(l[:i] + l[i+1:])  # error: unexpected indent
                    for x in p:
                                        r.append(l[i:i+1] + x)
                    return r         # error: inconsistent dedent
```

If you are lucky the errors will be detected by the parser but the code may run without error, but produce unexpected results!

while

The ***while*** loop executes a block of code as long as a condition is true.

```
count = 0
while count <= 10:
        count+=1
        print (count, end = " ")
        print ("Still in code block …", end = " ")
        print ("Still there …")

print ("Now outside code block")
```

Produces the same output as the ***for*** loop. The ***while*** loop is iterated until count is 11.

for or *while*

As they both achieve the same result, when do you use a ***for*** loop and when use a ***while***?

Generally, a ***for*** loop is used when you know the length of the data to be iterated and a ***while*** when you don't

For example, a program to display Fibonacci numbers:

A ***for*** loop would be used to display the first 10 Fibonacci numbers

A ***while*** loop would be used to display all Fibonacci numbers below 100

In the former the number is known, in the latter it isn't

Fibonacci Numbers

First 10 numbers:

```
first = 0
second = 1
print(0, end=" ")
print(1, end=" ")
fib = first + second
for i in range(8):
        print(fib, end=" ")
        first = second
        second = fib
        fib = first + second
```

```
0 1 1 2 3 5 8 13 21 34
```

Numbers below 100:

```
first = 0
second = 1
print(0, end=" ")
print(1, end=" ")
fib = first + second
while fib < 100:
        print (fib, end=" ")
        first = second
        second = fib
        fib = first + second
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

if Statement

The ***if*** statement is common to most programming languages, including Perl and Java.

It tests for a condition and acts upon the result of that condition.

If the condition is true, the block of code after the "***if*** condition" will be executed.

If it is false, the program will skip that block and will test for the next condition (if any).

Several conditions can be tested using ***elif***.

If all conditions are false, the block under the optional ***else*** will be executed.

if Structure

```
if condition1:
        code block1
elif condition2:
        code block2
else:
        code block3
```

if Example

A program that converts Centigrade to Fahrenheit:

```
faren = float(input("Enter a temperature")) * 9.0 / 5 + 32

print ("That converts to",faren)
if faren > 212:
        print ("That is steam“)
elif faren > 112:
        print ("That  is very hot water“)
elif faren > 32:
        print ("That is water“)
else:
        print ("That is ice“)

print ("Program completed“)
```

NOTE: The “input” command produces a screen prompt and the entered value is assigned to “faren”.

else if in Other Languages

The else if command has probably the most variability between languages:

Perl:

```
if(condition1){
        code block1;
        }
elsif (condition2){
        code block2;
        }
else{
        code block3;
        }
```

Java:

```
if(condition1){
        code block1;
        }
else if (condition2){
        code block2;
        }
else{
        code block3;
        }
```

Nested *if* Statement

As with other control structures ***if*** statements can be nested, with each ***else***/ ***elif*** referring to the ***if*** within the same code block:

```
if x == y:
        print (x, "and", y, "are equal“)
else:
        if x < y:
                print (x, "is less than", y)
        else:
                print (x, "is greater than", y)
```

Nested *for* Statement

```
onetoten = range(1,11)
onetofive = range(1,6)
for count in onetoten:
        print (count)
        for count2 in onetofive:
                print (count2)
```

Nested *while* Statement

```
count = 0
count2 = 0
while count <= 10:
        count+=1
        print (count)
        while count2 <= 10:
                count2+=1
                print (count2)
```

Mixing *for*/*while*/*if*

```
onetoten = range(1,11)
for count in onetoten:
        print (count)
        if count > 5:
                count2 = 10
                while count2 <= 15:
                        count2+=1
                        print (count2)
        else:
                count3 = 20
                while count3 <= 25:
                        count3+=1
                        print (count3)
```

Exiting a Control Structure

Although control structures will exit (or at least they should!) when complete it is possible to force an exit with the ***break*** command.

```
for variable in list:
        block1
        if variable has value:
                break
        block 2
```

The ***break*** will exit the ***for*** loop when the ***if*** condition is true regardless of whether the list has been completed.

Loop *break and continue* Commands

The ***break*** and ***continue*** commands give finer control over loops in Python. A ***continue*** statement jumps execution to the top of the loop, whilst a ***break*** statement finishes the loop prematurely.

```
for x in range(10):
        if x % 2 == 0:
                continue
        if x > 6:
                break
        print (x)
```

Outputs:

```
1
3
5
```

Loop *else* Command

Python ***for*** and ***while*** loops can also include an optional ***else*** clause. If it is present in a ***for*** loop, the block under the ***else*** clause is executed when the loop terminates through exhaustion of the list, but not when the loop is terminated by a ***break*** statement.

```
for variable in list:
        block1
else:
        block2
```

When present in a ***while*** loop, the block under the ***else*** clause is executed when the condition becomes false but not when the loop is terminated by a ***break*** statement.

```
while condition:
        block1
else:
        block2
```

Example of *break and* else

The following loop searches for prime numbers:

```
for n in range(2, 10):
        for x in range(2, n):
                if n % x == 0:
                        print (n, 'equals', x, '*', n/x)
                        break
        else:
                # loop fell through without finding a factor
                print (n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Lists

A Python ***list*** is, as the name suggests, a series of values.

Lists are common to other languages and often called ***arrays***.

The ***range*** command produces a ***list:***

```
range(1, 5)
```

Produces a list:

```
1, 2, 3, 4
```

Creating a *list*

A Python ***list*** is declared by using square brackets and separating the values by commas:

<name of list> = [<value>, <value>, <value>]

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
numbers = [1, 2, 3, 4]
```

Types can be mixed within a ***list***:

```
mixed =  [1, 2, ‘three’, 4, ‘five]
```

A ***list*** can even contain another ***list***

```
mixed =  [1, 2, ‘three’, 4, ‘five, [1, 2, 3, 4, 5], 6]
```

Manipulating a *list*

The elements in a ***list*** are stored in the order they are added and are indexed accordingly.

As with other programming languages the index starts at 0, so the first element in the ***list*** is at index 0. The index also runs from the end of the list with the last element being -1.

```
mixed =  [1, 2, 'three', 4, 'five, [1, 2, 3, 4, 5], 6]
```

Index 0 is 1

Index 2 is 'three'

Index 5 is a list, [1, 2, 3, 4, 5]

Lists are dynamic and can be added to, or elements deleted, and the list is resized accordingly.

List Operators

Elements of a list are accessed by the index in square brackets.

list[2]accesses the element at index 2

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
month = months[2]
```

Assigns ‘Mar’ to variable month

In the same way that elements can be accessed they can also be changed;

```
month[3] = “Aug”
```

Now months is:

```
['Jan', 'Feb', ‘Mar', ‘Aug', 'May', 'Jun']
```

Modifying a *list*

You can add items to an existing sequence.

The **append** method adds a single item to the end of the list:

```
months.append('Jul')
```

The **extend** method adds items from another list to the end:

```
months2 = ['Aug', 'Sep', 'Oct']
months.extend(months2)
```

Insert inserts an item at a given index, and moves the remaining items to the right.

```
list.insert(index, item)
months.insert(3, 'Dec')
```

Modifying a *list* (cont…)

You can also remove items from a list.

The **del** statement can be used to remove an individual item, or to remove all items identified by a slice. A slice identifies a starting and ending index.

```
del months[2]
```

The **pop** method removes an individual item and returns it. With no index specified it removes the last item in the list.

```
month = months.pop() # last item
month = months.pop(0) # first item
```

The **del** statement and the **pop** method does pretty much the same thing, except that **pop** returns the removed item.

The **remove** method searches for an item, and removes the first matching item from the list.

```
months.remove('Jul')
```

Other *list* Methods

The order of the list can be reversed or sorted:

```
months.reverse()
months.sort()
```

The length of a list can be returned:

```
length = len(months)
```

The presence of an item can be checked:

```
if 'Jul' in months:
        print ("In list")

if 'Jul' not in months:
        print ("Not in list")
```

The index position of the first occurrence of an item in a list can be returned:

```
index = months.index('Jul')
```

List Slices

Slices are used to return part of a list.

The ***slice*** operator is in the form ***list[first_index:following_index]***.

The ***slice*** goes from the ***first_index*** to the index before the ***following_index***.

If the first index is not specified the beginning of the list is assumed.

If the following index is not specified the whole rest of the list is assumed.

You can use both types of indexing:

```
list = ['zero','one','two','three','four','five']


list[0:3]  ->  ['zero','one','two']
list[-4:-2]  ->  ['two','three','four',]
list[-5:6]  ->  ['one','two','three','four','five']
list[3:]  ->  ['three','four','five']
list[:3]  ->  ['zero','one','two']
```

List Slices (cont…)

An optional addition to the slice function is the ***step.*** This determines how many items in the list are skipped:

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

list[2:8:2]  ->  [3, 5, 7]
```

The default value for step is 1 so it is usually not necessary for it to be included.

The interesting use of step is when it is negative:

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

list[8:2:-2]  ->  [9, 7, 5]
```

Or:

```
list[::-1]  ->  [10, 9, 8, 8, 7, 6, 5, 4, 3, 2, 1]
```

The list is reversed! Not necessary as there is already a reverse function for lists but this has another use.

Copying Lists

There are several ways to make a copy of a list.
The simplest that works most of the time is the slice operator since it always makes a new list even if it is a slice of a whole list:

```
a = [1,2,3]
b = a[:]
b[1] = 10

a is now [1, 2, 3]
b is  [1, 10, 3]
```

Just using "***b = a***" would make "***b***" a reference to "***a***" so any change to "***a***" would also change "***b***".

Taking the ***slice[:]*** creates a new copy of the list. However, as expected a list can contain an inner list as an element but ***slice[:]*** only copies the outer list. Any sublist inside is still a reference to the sublist in the original list. Therefore, when the list contains lists the inner lists have to be copied as well. You could do that manually but Python already contains a module to do it. This is covered later.

Tuples

Tuples are like ***lists*** but they can not be modified. Items have to be enclosed by parentheses instead of square brackets to create a ***tuple*** instead of a list.

In general all that can be done using ***tuples*** can be done with ***lists***, but sometimes it is more secure to prevent internal changes.

```
list = ['zero','one','two','three','four','five']

tuple = ('zero','one','two','three','four','five‘)
```

Back To Strings

With Python a string is essentially a list of chars and so can often be manipulated in the same way as a list. This means that some operators that work on lists can also be used on strings, particularly slice:

For example to substring or get string length:

```
name = “bioinformatics”
part = name[2:8]
print (part, “has length”, len(part))
```

-> oinfor has length 6

Some More String Methods

```
name = ‘Bioinformatics'
if name.startswith(‘Bio'):
        print ('Yes, the string starts with “Bio”’)
if ‘form' in name:
        print ('Yes, it contains the string “form"‘)
```

Outputs:

Yes, the string starts with “Bio"
Yes, it contains the string “form"

Some More String Methods (cont…)

capitalize() Return a copy of the string with only its first character capitalized.

count(*sub*[, *start*[, *end*]]) Return the number of occurrences of substring *sub* in string S[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

find(*sub*[, *start*[, *end*]]) Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

lower() / **upper**() Return a copy of the string converted to lower/upper case.

replace(*old, new*[, *count*]) Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

There are many more …

Reversing a String

Strings are essentially lists and lists can be reversed with the ***reverse*** function.

However, this does not work with strings.

Python does not provide a built in string reversal function.

However, there is the slice option to reverse a list, and hence a string:

```
s = “abcde”
s = s[::-1]
```

The value of s now “edcba”

Back to Loops

As mentioned previously, the range used in ***for*** loops is a ***list***.

Any ***list*** can therefore be used:

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']

for month in months:
        print ("The month is:", month)
```

Dictionaries

Dictionaries are the Python version of the ***hash***, or ***associative array***.

Dictionaries are like ***lists*** but instead of having numbers as their index they can have any value as an index (key) associated with the array element (value).

Dictionaries associate a key with a value, an example being associating codons with their amino acids:

‘ttt’ > ‘F’
‘tta’ > ‘L’
etc

Dictionaries enable this type of data to be stored and handled.

Creating a *Dictionary*

A ***dictionary*** is created in a similar manner to a an array, but with key/value pairs and using curly braces:

```
codons = {‘ttt’:’F’, ‘tta’:’L’, ‘gga’:’G’}
```
Note curly braces

Or items can be added directly:

```
codons[‘aac’] = ‘N’
```
Note square braces

Using a *Dictionary*

A key can be searched for:

```
if codon in codons:     # 2.7 version is codons.has_key('aac'):
```

All ***keys*** or ***values*** can be retrieved as a list:

```
keys = codons.keys()
for x in keys:
        print (x)
        print (codons[x])  # Prints the value for this key
```

Or:

```
values = codons.values()
for x in values:
        print x
```

Using a *Dictionary* (cont…)

Removing ***key:value*** pairs:

You can use **del** to remove a key:value pair from a dictionary.

```
del codons['aac']
```

File Handling

Basic file handling is very straightforward.

To write to a file:

```
out_file = open("test.txt","w")  # “a” to append
out_file.write("This Text is going to out file\nSome more text\n")
out_file.close()
```

To read a file:

```
in_file = open("test.txt","r")  # “r” is optional as it is the default
text = in_file.read()
in_file.close()
print (text)
```

Reading a File

Programming languages all provide methods for reading files

The syntax varies greatly (simple for Python and Perl, more complex for Java)

Once the file is open for reading they provide methods to read the file one line at a time

Reading the Whole File

The readlines command reads the file one line at a time but an alternative is to read the entire file contents with read:

```
in_file = open(filename)
contents = in_file.read()
print (contents)

in_file.close()
```

Using for Loop

The file contents can also be read into a list and iterated over in a ***for*** loop:

```
in_file = open(filename)
for line in in_file.readlines():
        print (line)
in_file.close()
```

It has also possible to treat the file as a list so you don't need to use readlines() inside a ***for*** loop, you just iterate over the file:

```
in_file = open(filename)
for line in in_file:
        print (line)
in_file.close()
```

Using “with”

Another way of working with files is the **with** statement.

It is good practice to use this statement.

With the **with** statement you get better syntax and exceptions handling.

It will also automatically close the file so the **with** statement provides a way for ensuring that a clean-up is always used.

```
with open(filename) as file:
```

You can of course also loop over the file as before:

```
with open("newfile.txt") as f:
        for line in f:
            print (line)
```

Note there is no need to close the file, it is done automatically.

Removing the Newline Character

Python provides a string method called strip() which will remove *whitespace*, including newlines, from both ends of a string. It also has variants which can strip one end only called rstrip and lstrip too. The rstrip method can be used to remove newlines:

```
in_file = open(filename)
for line in in_file:
        strip_line = line.rstrip()
        print (strip_line)
in_file.close()
```

Or more simply:

```
in_file = open(filename)
for line in in_file:
        print (line.rstrip())
in_file.close()
```

NOTE: An alternative is line = line[:-1]

Writing Embedded Variables to a File

When writing text to a file it is constructed as a string, which means any variables need to be appended to quoted text with a plus sign. It also means that non string variables, such as integers, need to be cast as strings - **str(int value)**:

```
name = "Dave"
age = 20

out_file = open("test.txt","w")

out_file.write("Student: Name = " + name + " Age = " + str(age) = "\n")

 out_file.close()
```

Note that spaces are not automatically added after each quoted string and a newline is not automatically added.

If required both need to be included in the write statement.

A Simple Program

A common task in bioinformatics is using a scripting language, such as Python, to also parse the output from an analysis program, such as BLAST, and produce a file of edited data.

Scripting languages can be used to rapidly write code to read files, manipulate the text and produce an output.

An example might be to read a fasta file of nucleotide sequence and produce the possible amino acid sequence. In other words, translate the sequence.

This will demonstrate the use of file handling, control structures, string (list) manipulation and dictionaries.

Input Files

For the example we will have 2 input files.

The fasta sequence:

>BF246290
ggcgtcgtagtctcctgcagcgtctggggtttccgttgcagtcctcggaaccaggacctc
ggcgtggcctagcgagttatggcgacgaaggccgtgtgcgtgctgaagggcgacgg
cccagtgcagggctatcatcaattcgagcagaaggaaagtaatggcaccagtgaag
…
cacagatggtgtgccgatgtgtctatggaacgattctgtgatctcactctcaggagacca
tggcatcatgtggccgcacaactgtggtccatgaaaaagcaagatgactgtgggcca
ggg

NOTE: This is not one complete sequence, there are line breaks

A file of codon translations:

ttt
F
ttc
F
tta
L
ttg
L etc

Storing the Sequence

The first step is to read the fasta sequence from file into a string:

```
in_file = open('fasta_file')  # Open the file

seq_list = in_file.readlines()  # Read the file to a list

seq_name = seq_list.pop(0)  # Remove the sequence name

seq = seq_list.pop(0)  # Initialise the sequence

seq = seq.rstrip()  # Remove the newline character

seq = seq.lower()  # Lower case the sequence

for line in seq_list:  #  Append the rest of the sequence to seq
        seq += line.rstrip().lower()
in_file.close()
```

NOTE: Methods can be appended to each other - seq += line.rstrip().lower()

Building the Dictionary

The next step is to build the dictionary using the codons file:

```
in_file = open('codons_file')  # Open the file

codons_list = in_file.readlines()  # Read the file to a list

codons = {}  # Initialise the dictionary

for count in range(0, len(codons_list), 2):  # Work through the list
        key = codons_list[count].rstrip()   # Get the codon
        key = key.lower()
        value = codons_list[count+1].rstrip()  # Get the aa, on next line
        value = value.lower()
        codons[key] = value  # Add to dictionary

in_file.close()
```

Translating the Sequence

Now we have the sequence and codon translations we create the amino acid sequence:

```
for count in range(0, len(seq), 3):  # Work through the list
        codon = seq[count:count+3]   # Get 3 nucleotides
        aa = codons[codon]  # Get the associated aa
        print (aa, end="")
```

Complete Program

Putting it all together, with a few short cuts:

```
in_file = open('fasta_file.txt')
seq_list = in_file.readlines()
seq_name = seq_list.pop(0)
seq = seq_list.pop(0).rstrip().lower()

for line in seq_list:
        seq += line.rstrip().lower()
in_file.close()

in_file = open('codons.txt')
codons_list = in_file.readlines()

codons = {}  # Initialise the dictionary
for count in range(0, len(codons_list), 2):
        codons[codons_list[count].rstrip().lower()] = codons_list[count+1].rstrip().lower()

in_file.close()

for count in range(0, len(seq), 3):
        codon = seq[count:count+3]
        if codon in codons:
                aa = codons[codon]
        else:
                aa = '-'
        print (aa, end="")
```

Modules

Modules, or libraries, are common to most programming languages, including Perl, C++ and Java.

Modules provide a set of code to provide particular functions that can be included in your own code.

They are essentially programs with functions that can be called from your own program.

The previous program that translates nucleotide sequences could be converted into a module.

The module could then be used in another program that could “use” it simply by providing the file containing the nucleotide sequence and calling the function to translate it.

Using a Translate Module

Assuming the module was called ***trans*** and the code file was called ***seq.fas*** the code could be something like:

```
import trans

trans.translate(seq.fas)
```

The module code would then translate the sequence and print out the results.

In a real module it would probably be better to return the translated code so it could be printed out or modified in the calling program.

The important point is that the code that actually does the translation is hidden from the calling program and there is no need to see or understand it.

Using Modules

As expected Python provides a large library of modules, including Biopython and PyCogent, and are imported with the "***import***" command.

The most commonly used is probably the ***sys*** module which contains system-specific functionality. This is required for functions that are often included by default in other languages.

One example being the use of command line arguments. These are the arguments that can be passed to a program when you run it, for example BLAST:

```
blastall –p blastn –i seq.fas –o seq.blast
```

Python Command Line Arguments

An example is:

```
python test.py arg1 arg2
```

Then the script test.py could print out the command line arguments, using the system module:

```
import sys

print “Argument 1 is”, sys.argv[1]
print “Argument 2 is”, sys.argv[2]
```

Prints:

```
Argument 1 is arg1
Argument 2 is arg2
```

NOTE: The first element of the sys.argv list is actually the script name, in this case test.py. The first command line argument is the second in the list, argv[1].

Command Line Arguments (cont…)

You can check for the correct number of arguments:

```
if len(sys.argv) < 2:
        print ‘Argument missing.'
        sys.exit()  # Use the sys exit method to quit
```

To retrieve arguments:

```
arg1 = sys.argv[1]
```

Command Line Arguments (cont…)

When using the ***sys argv*** command have to type ***sys.argv*** every time it is needed.

However, the ***argv*** variable can be directly imported into the program, thus avoiding the need to type the *sys* every time for it.

To enable this use the ***from sys import argv*** statement:

```
from sys import argv

arg1 = argv[1]
arg2 = argv[2]
```

If you want to import all the names used in the sys module, then you can use the ***from sys import** *** statement.

This works for any module.

Back to Lists

As mentioned previously with lists, taking the slice [:] creates a new copy of the list.

However, it only copies the outer list.

Any sublist inside is still a references to the sublist in the original list.

Therefore, when the list contains lists the inner lists have to be copied as well.

You could do that manually but Python already contains a module to do it.

You use the ***deepcopy*** function of the ***copy*** module:

deepcopy

To use the ***deepcopy*** module:

```
import copy

a = [[1,2,3],[4,5,6]]
b = a[:]
c = copy.deepcopy(a)
b[0][1] = 10  # Will change a too
c[1][1] = 12  # Will not change a
print “List a”, a
print “List b”, b
print “List c”, c
```

Outputs

```
List a [[1, 10, 3], [4, 5, 6]]
List b [[1, 10, 3], [4, 5, 6]]
List c [[1, 2, 3], [4, 12, 6]]
```

System Calls

One of the main uses of scripting languages such as Python is to connect together other programs (pipeline), feeding the output, possibly parsed, as the input to another program.

In order to do this you need to run external programs from within the Python script.

This is called a system call and the ability to do this is present in other languages, such as Perl and Java, but the syntax varies greatly

System Calls in Python

System calls in Python require the *os* module:

```
import os

cmd = ‘blastall –p blastn –i test.fasta –o test.blast'
os.system(cmd)
```

This will run the BLAST blastall command but will not capture the output. That can be achieved with the *os* module *popen* command.

```
import os

cmd = ‘blastall –p blastn –i test.fasta'
blast_output = os.popen(cmd).read()
```

NOTE: ***os.popen(cmd)*** by itself will return a list of the output lines. This can then be parsed in a ***for*** loop

Functions

Often when writing a program there are times when you want to use the same piece of code multiple times.

An example is the sequence translation program. What if you wanted to translate multiple sequence?

A flaw in the program is also that is only translates the sequence in one reading frame. It should translate it in all 6 reading frames. That is a fairly straightforward modification of the code but writing the final translation part six times is very inefficient.

This is where functions are used (or subroutines or methods, depending on the language).

A function is a block of code that can be used multiple times to perform a particular piece of work.

Simple Function

```
def hello():
        print "Hello World!"
```

A function is defined by the keyword def followed by the function name. Note the brackets following the name, as functions can optionally take arguments.

The structure of a function is like Python control methods, the function name ending in a colon and the scope of the function identified by indented code.

Using a Function

A function has to be declared in the program before it can be used.

The function is called by simply using its name, and including arguments if required.

The code within the function is then called:

```
def hello():
        print "Hello World!"


hello()
hello()
```

Outputs:

```
Hello World!
Hello World!
```

Passing Arguments to a Function

Arguments can be passed that can then be used within the function:

```
def hello(name):
        print ("Hello“, name, “!”)


hello(“Lenka”)
hello(“Martin”)
```

Outputs:

```
Hello Lenka!
Hello Martin!
```

Passing Multiple Arguments

Any number of arguments can be passed:

```
def hello(name1, name2):
        print ("Hello“, name1, “and”, name2)


hello(“Lenka”, “Martin”)
```

Outputs:

```
Hello Lenka and Martin
```

Returning Values

Functions can also return data:

```
def add(num1, num2):
        num3 = num1 + num2
        return num3


num = add(3, 4)
print (num)
```

Outputs:

```
7
```

Returning Values (cont…)

The use of functions can be abbreviated:

```
def add(num1, num2):
        return num1 + num2


print (add(3, 4))
```

Returning Multiple Values

```
def calc(num1, num2):
        a = num1 + num2
        b = num1 * num2
        c = num1 – num2
        return a,b,c


d, e, f = calc(10, 5)
print (d, e, f)
```

Outputs:

```
15 50 5
```

The function is actually creating a tuple.

Returning Multiple Values (cont…)

Although the previous version actually returns a tuple it is possible to explicitly return one, or a list:

```
def calcTuple(num1, num2):
        a = num1 + num2
        b = num1 * num2
        c = num1 – num2
        return (a,b,c)
```

```
def calcList(num1, num2):
        a = num1 + num2
        b = num1 * num2
        c = num1 – num2
        return [a,b,c]
```

Returning Multiple Values (cont…)

The functions return a tuple or list which can be assigned to another tuple or list or to the individual values:

```
d, e, f = calcTuple(10, 5)  #  Assigns 15, 50, 5 to d, e, f

g, h, i = calcList(10, 5)  #  Assigns 15, 50, 5 to g, h, i

j = calcTuple(10, 5)  # Assigns a tuple of 15, 50, 5 to j

k = calcList(10, 5) # Assigns a list of 15, 50, 5 to k
```

```
k = calcList(10, 5)
for num in k:
        print (num)
```

Outputs:

```
15

50

5
```

Passing Arguments (cont…)

Any data type, or object, can be passed to a Python function:

```
def calcSum(list1):
        total = 0
        for num in list1:
                total += num
        return total


sumlist = (1, 2, 3, 4, 5)
print (calcSum(sumlist))
```

Outputs:

```
15
```

List/Dictionary Arguments

Lists and dictionaries are passed by reference to functions so care needs to be taken:

```
num = 8
arr = [1, 2, 3, 4, 5]

def testDef(n, a):
        n = 1
        a[2] = 6

print "1 Num", num, "Arr", arr
testDef(num, arr)
print "2 Num", num, "Arr", arr
```

Output;

```
1 Num 8 Arr [1, 2, 3, 4, 5]
2 Num 8 Arr [1, 2, 6, 4, 5]
```

If in doubt copy (preferably *deepcopy)* the list/dictionary before passing it to the function.

Other Argument Options

There are many other options for passing arguments to a function, including:

- Passing function arguments by keywords
- Default values of parameters
- Variable number of parameters

These are covered in more detail in the tutorial.

Variable Scope

The visibility, or scope, of a variable is something that has to be considered in all languages.

Scope refers to where within a program a declared variable can be accessed.

Scope is enforced far more in compiled languages than scripting languages.

Consider in Java:

```
for(int i = 0; i <= 10; i++){
            System.out.prtintln(“”+i);
            }
 System.out.prtintln(“”+i);
```

This would produce an error and the code would fail to compile.

The reason for this is that the variable i is declared within the for loop and so can only be accessed within that loop. The variable scope is restricted to the code block it is declared in.

Variable Scope in Python

In Python all variables have global scope, that is they can be accessed from anywhere in the program, with one exception.

Variables within functions only have scope within that function:

```
num3 = 20
def mul(num1, num2):
        num3 = num1 * num2
        return num3


print (mul(3, 4))
print (num3)
```

Output:

```
12
20
```

Global Variables

If required it is possible to make to make a variable declared within a function globally accessible by using the ***global*** keyword:

```
num3 = 20
def mul(num1, num2):
        global num3   # Declare num3 to be global
        num3 = num1 * num2
        return num3


print (mul(3, 4))
print (num3)
```

Output:

```
12
12
```

Regular Expressions

A regular expression (regex or regexp for short) is a special text string for describing a search pattern.

You are probably familiar with wildcard notations such as *.txt to find all text files in a file manager.

The regex equivalent is .*\.txt$.

That is any number (0 or more) of characters - `.*`

Followed by a full stop - `\.`

The full stop is escaped to distinguish it from the special character used previously.

Followed by the suffix that must be on the end of the string - `txt$`

When starting you should keep them simple and verbose so they are easy to understand

Regular Expressions (cont…)

Regular expressions are an incredibly powerful tool

They are particularly well integrated into Perl

In Perl they are built in but in Python provided by a module

Regular expressions can be difficult to understand and learn

For example, to match full length eukaryotic mRNA sequences:

```
^ATG[ATGC]{30,1000}A{5,10}$
```

Matches:

An ATG start codon at the beginning of the sequence
Followed by between 30 and 1000 bases which can be A, T, G or C
Finally, a poly-A tail of between 5 and 10 bases at the end of the sequence

Regular Expressions (cont…)

In Python regular expressions are provided by the regular expression (***re***) module.

The ***re*** module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

To compile and use a ***re***:

```
import re

p = re.compile('\d+')   # Match any digits
m = p.match( '1 to 45' )  # Apply the re
if m:
        print ('Match found')
else:
        print ('No match')
```

Output:

```
Match found
```

Match/Search

The ***match*** method only checks if the RE matches at the start of a string.

The ***search*** method matches anywhere within the string.

```
p = re.compile(‘\d+')   # Match any digits
m = p.match( ‘Try 1 to 45' )  # Apply the re
if m:
        print ('Match found‘)
else:
        print ('No match')


m = p.search( ‘Try 1 to 45' )  # Apply the re
if m:
        print (‘Search found‘)
else:
        print ('No search found')
```

Output:

```
No match
Search found
```

RE Methods/Attributes

The re match object has methods and attributes that can be used to return information about the matching string. The most important ones are:

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

```
import re
p = re.compile(r‘inform')
m = p.search(‘bioinformatics')
grp =  m.group()
start = m.start()
end = m.end()
span = m.span()
print (“Match is: “, grp)
print (“Start is”, start, “end is”, end)
print (“Span is”, span)
```

Output:

```
Match is: inform
Start is 3 end is 9
Span is (3, 9)
```

NOTE: As a string is being matched an “r” is added to the compiler to match raw strings. This should probably be used as a general rule and certainly if in doubt. For example, without it \b will be converted to a backspace

Subgroups

When matching a pattern in a string you may only want to return part of the match

```
ACCESSION       BF246290
```

To return just the accession number and not the entire line

```
p = re.compile(r“^ACCESSION\s(\w+)”)
m = p.search(line_from_file)

print m.group(1)
```

```
BF246290
```

An alternative is to split the line into a list and print index 1

Subgroups (cont…)

Multiple matches can be returned

line = “LOCUS BF246290 857 bp mRNA EST 14-NOV-2000”

```
p = re.compile(r“^LOCUS\s+\w+\s+(\d+\s+bp)\s+(\w+)”)
m = p.search(line)

print “Length is“,m.group(1),” and type is,”m.group(2)
```

```
Length is 857 bp and type is mRNA
```

Subgroups (cont…)

You can also nest the matched groups

```
p = re.compile(r'(a(b)c)d')
m = p.match('abcd')
m.group(0) is 'abcd‘
m.group(1) is 'abc‘
m.group(2) is 'b'
```

Count the opening parenthesis characters from left to right

Multiple Matches

Two ***re*** methods return all of the matches for a pattern. ***findall()*** returns a list of matching strings:

```
p = re.compile(r'\d+')
p.findall('The first is 12, next 7 and 2 is last')
```

Output:

```
['12', '7', '2']
```

As it is a list:

```
for match in p.findall('The first is 12, next 7 and 2 is last') :
        print (match)
```

Output:

```
12
7
2
```

Module-Level Functions

You don't have to produce a ***re*** object and call its methods; the ***re*** module also provides top-level functions called ***match()***, ***search()***, ***sub()***, etc.

These functions take the same arguments as the corresponding object method, with the RE string added as the first argument, and still return either None or an object instance.

```
if re.search(r'inform', 'bioinformatics'):
        print ("Found")
else:
        print ("Not found")
```

Output:

```
Found
```

Which RE Method?

The choice of which method to choose is dependant on 2 requirements:

1. Is the re being used only once in the code?

   The module-level functions would be suitable

2. Is the re going to be used multiple times?

   Creating an re object may be more efficient

The actual choice comes down to personal programming style preferences and there is probably not a definitively correct choice.

Modifying Strings

Python regular expressions can also be used to modify strings. The ***re*** module provides the following methods:

Method/Attribute	**Purpose**
split()	Split the string into a list, splitting it wherever the RE matches
sub()	Find all substrings where the RE matches, and replace them with a different string
subn()	Does the same thing as sub(), but returns the new string and the number of replacements

split

Splits a string apart wherever the ***re*** matches. The maximum number of splits can also be set with the remainder of the string being the last element returned:

split(*string* [*, maxsplit = 0*])

```
p = re.compile(r'\s+')
p.split('This is a test of the split method.')
```

Output:

```
['This', 'is', 'a', 'test', ‘of', ‘the’, ‘split', ‘method']
```

```
p.split('This is a test of the split method.‘, 3)
```

Output

```
['This', 'is', 'a', ‘test of the split method.']
```

String split Method

As previously mentioned, strings have many inbuilt methods, including a split method:

```
str.split([delim[, maxsplit]])
```

If no delimiter is given then by default the string is split on whitespace:

```
s = '1  2  3‘
s.split()
returns ['1', '2', '3']
```

With delimiter:

```
s = '1XX2XX3‘
s.split('XX')
returns ['1', '2', '3']
```

Search and Replace

The ***sub()*** method finds all the matches for a pattern, and replaces them with a different string.

sub(*replacement, string*[*, count = 0*])

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the ***re*** in ***string*** by the replacement ***replacement***.

If the pattern isn't found, ***string*** is returned unchanged.

The optional argument ***count*** is the maximum number of pattern occurrences to be replaced; ***count*** must be a non-negative integer. The default value of 0 means to replace all occurrences.

sub()

```
p = re.compile( r'(blue|white|red)')
p.sub( 'colour', 'blue socks and red shoes')
```

Output:

```
'colour socks and colour shoes'
```

```
p.sub( 'colour', 'blue socks and red shoes', count=1)
```

Output:

```
'colour socks and red shoes'
```

subn()

The subn() method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
p = re.compile(r'(blue|white|red)')
p.subn('colour', 'blue socks and red shoes')
```

Output:

```
('colour socks and colour shoes', 2)
```

```
p.subn('colour', 'no colours at all')
```

Output:

```
('no colours at all', 0)
```

String sub/replace Method

Strings also include a built in replace method:

```
str.replace(old, new[, count])
```

```
s = 'blue socks and red shoes‘
r = s.replace('red', 'colour')
print (r)
```

Prints:

```
'blue socks and colour shoes'
```

Naming Considerations

In Python the names for all kinds of things - numbers, strings, functions, modules - are treated the same way. This is different to other languages such as Java where the variable foo and the class foo are considered two different things.

This can be demonstrated with an example:

```
spoon = 2 + 2
print (spoon)

def spoon():
        return 3

print (spoon())
spoon = 'foo'
print (spoon)
```

Output:

```
4
3
foo
```

Naming Errors

Scripting languages offer a great deal of flexibility but as a result also have drawbacks.

One of the potential problems is created by the loose control of variable declaration.

Consider:

```
number1 = 20
number2 = 5
number3 = number1 * number2
print (number3)
numbr3 = 30
print (number3)
```

Output:

```
100
100   # Not 30 as intended
```

Error Handling

One of the problems with any programming language is handling errors and preventing the program from crashing. Take the following example:

```
print ("Type Control C or -1 to exit")
number = 1
while number != -1:
        number = int(input("Enter a number: "))
        print ("You entered: ", number)
```

The program will prompt for a number and print it out. It will continue prompting until the exit condition is reached.

It takes a string as input and converts it to an integer.

There is a problem with the previous code in that what would happen if text other than a number was entered, for example “2d”. Assuming the script was called script.py it would produce the following error message and exit:

```
Traceback (innermost last):
        File “script.py", line 4, in ?
                number = int(input("Enter a number: "))
ValueError: invalid literal for int(): 2d
```

It would be better to be able to catch errors like these and provide an opportunity to correct them so that the program could continue to run. Alternatively, at least exit more cleanly with an appropriate error message.

Python provides a method to do this.

Try/Except

```
print ("Type Control C or -1 to exit“)
number = 1
while number != -1:
        try:
                number = int(input("Enter a number: "))
                print ("You entered: ", number)
        except ValueError:
                print ("That was not a number.“)
```

Now when something like “2d” is entered it prints “That was not a number” and continues without exiting.

The previous example will only handle errors with the number format, any other errors will not be caught and the script will still crash.

It is possible to catch multiple possible errors:

```
try:
        number = int(input("Enter a number: "))
        print ("You entered: ",number)
        ans = 10/number
        print (“10 divided by “, number, “=“, ans)
 except ZeroDivisionError:
        print (“Cannot divide by 0!”)
except ValueError:
        print ("That was not a number.")
```

Finally

The finally block can be added to a ***try*** ... ***except*** statement and will be executed whether or not an exception is raised.

One reason you may want to use this is to ensure a file is closed cleanly. For example, in a modified version of the previous script that reads numbers from a file:

```
in_file = file("nums.txt")
try:
        for line in in_file:
                number = int(line.rstrip())
                print ("Number is: ", number)
                ans = 10/number
                print ("10 divided by ", number, "=", ans)
except ZeroDivisionError:
        print ("Cannot divide by 0!“)
except ValueError:
        print ("That was not a number.“)
finally:
      in_file.close()
```

Interactive Mode

Python can be run on the command line using it's interactive mode. Just type ***Python*** and interactive mode will be started with a prompt, which on codon is:

```
Python 2.3.3 (#1, Nov 22 2005, 01:28:01) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> sign is an indication that Python is waiting for input.

Interactive Mode Examples

```
>>> print “Hello World”
Hello world
>>>
```

```
>>> import re
>>> p = re.compile(r'inform', re.I)
>>> m = p.search('Bioinformatics')
>>> m.group()
'INFORM'
>>> m.start(), m.end()
(3, 9)
>>> m.span()
(3, 9)
>>>
```

Conclusions

These lectures have covered the main aspects of Python

They have provided enough for you to now write Python scripts but there is more to the language, including Object Oriented code options

The best way to develop coding skills is to write more code!

Using and editing existing scripts or code snippets can prove a productive way to develop skills

Writing the code is only one aspect of programming – the main challenge is developing the algorithm

Use the tutorials provided to gain an understanding of the language

Read the related web sites and tutorials where available