Python

Python is a general-purpose high-level programming language, released in 1991 by Guido Van Rossum (currently employed by Google). Its name refers to the television series Monty Python's Flying Circus.

Conceptually and structurally similar to Perl; syntax is tidier and it is more cleanly object oriented.
 Used for a variety of applications, from scripting to web programming (Abaqus, Maya, MotionBuilder, Softimage, Cinema 4D, BodyPaint 3D, modo, Blender, GIMP, Inkscape, Scribus, Paint Shop Pro, etc)
 Adopted by Google, Yahoo!, Youtube, CERN, NASA, Red Hat....
 one of the scripting languages in Google Docs.
 Already included in Linux, NetBSD, OpenBSD, Mac OS X.
 Free and open source
 Stable (~20 years old) but continuingly evolving
 Very portable: it runs on almost every operating system (Unix, Windows, Mac, BeOS, Win/CE, DOS, OS/2, Amiga, VMS, Cray, …)
 Easy to learn: close to pseudocode
 Easy to read: no brackets, just indentation
 Easy to debug
 Minimalistic syntax
 Powerful, vast collection of libraries (batteries included)
 Widely used in bioinformatics
 Object oriented
 GUI: Tkinter, Qt, GTK, win32, wxWidgets, Cairo, ...

Main drawback: speed
 Interpreted language - speed comparable to perl, java, etc.
 Programs run slower than low-level languages, but they are faster to write
 programmer effort versus computer effort
 Solution: modules can also be written in C

gotcha
 There are TWO versions of Python in common use: 2.7 and 3
 syntax is different though not substantially, and Python 3 is largely but not completely back compatible

Basic Syntax (thanks to Loris Mularoni for notes that parts of this are heavily based on)

start python from the shell:

```
$python
Python 2.7.10rc1 (v2.7.10rc1:80ccce248ba2, May 10 2015, 11:32:08)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To exit, type exit() or ^D

# denotes a comment; anything following this symbol is not executed

```
>>> # this is just a comment, not a command
```

It is HIGHLY recommended to heavily comment your code! the minimum level of commentary needed to ensure that you will be able to run your program again in 6 months is roughly one line of comments per 5 lines of code.

Variables
A variable is just a name for an object. In Python, the variables themselves can be attached to objects of any type (number, character etc) without recasting.

```
>>> a = 1
>>> a
1
>>> b = "hi"
>>> b
'hi'
>>> b = 2
>>> b
2
>>> c = 10
>>> a==c
False
>>> a < c
True
>>> a <= c
True
>>> b = “hi"
```

Are a and b the same? (can we compare them)?
If I try a=b I’ll just set the value of a to the value of b.
== is the comparison operator.
To test whether the two variables are not equal, use the != operator.
Other comparison operators (>, <, >=, <= etc) are more familiar.

```
>>> a==b
False
>>> a!=b
True
>>> a < b
True
>>> a + c
11
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Variable types:
number
string (character)
list
tuple
dictionary

Numbers
There are different types of numbers:

integer (32 bit)	int()
floating point (unlimited precision)	float()
long	long()
complex	complex()

and you do need to know about these:
```
>>> 3+4
7
>>> 3/4
0
>>> 4/2
2
>>> 4/3
1
```

The default number type is "int." If you want your arithmetic to use more precision, one or more numbers must be "float" or "long."

```
>>> float(4)/3
1.3333333333333333
>>> float(4/3)
1.0
5.0
>>> int(5.4)
5
```

To check the type of your object:
```
>>> a = 10
>>> type(a)
<type 'int'>
>>> b = "hi"
>>> type(b)
<type ‘str'>
```

dir is a very useful command: it lists the functions that apply to the type of object you pass to it.
```
>>> dir(b)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', ‘zfill']
```

You can also use help() to get details about functions that can be used with a object:

```
>>>help(a)
```

Strings

A string is a sequence of characters (any character, including tabs and spaces).

```
>>> seq1 = "accgt"
>>> seq2 = "cccat"
>>> seq1 == seq2
False
>>> seq1 != seq2
True
```

Let’s look at the first character of seq1:

```
>>> seq1[1]
‘c'
```

wait a minute . . .
In Python (and most languages, but not R), counting starts at 0:

```
>>> seq1[0]
‘a'
```

Take a slice out of seq1 (the string that is returned will include everything starting at the first position requested, up to but not including the last position requested)

```
>>> seq1[0:2]
'ac'
>>> seq1[0:3]
'acc'
>>> seq1[1:3]
'cc'
>>> seq1[2]==seq2[2]
True
>>> len(seq1)
5
>>> seq1[5]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> seq1[4]
't'
>>> seq1[-1]
't'
```

Python is a fantastic language for string operations and has many built-in functions.

```
>>> "g" in seq1
True
>>> "g" in seq2
False
>>> seq1.count("c")
2
>>> seq1.find("c")
1
>>> seq2.find("at")
3
>>> seq2.find("g")
-1
>>> seq1.upper()
'ACCGT'
>>> seq1.replace("a", "T")
'Tccgt'
>>> seq1.upper()
'TCCGT'
>>> seq1.lower()
'tccgt'
```

Exercise:
Write a series of commands that transform the rna sequence "UUgGAagaGcuuACUUag" to DNA and then calculate its GC content.

Tips:
  Assign the sequence to a variable
  Make all the nucleotides uppercase (or lowercase)
  Replace all the 'U' with 'T' (or 'u' with 't' if you made the string lowercase)
  Count the number of 'C' and 'G' and divide by the length of the sequence

Lists
Lists are packages of objects that can be any type (including other lists). They are created using the [ ] operator.

```
>>> practicelist = [42, "a few letters", "polka dot", 7, -1.33]
>>> print practicelist
[42, 'a few letters', 'polka dot', 7, -1.33]
```

```
>>> practicelist[1]
'a few letters'
>>> practicelist[0]
42
>>> len(practicelist)
5
```

You can slice a list just as you select a piece of a string:

```
>>> practicelist[1:2]
['a few letters']
>>> practicelist[1:3]
['a few letters', 'polka dot']
```

The split function creates a list that contains the pieces of the original object, that were separated by the chosen delimiter:

```
>>> seq = "aggNattgNagggtgNat"
>>> seq.split("N")
['agg', 'attg', 'agggtg', 'at']
>>> newlist = seq.split(“N”)
```

The list function splits a string into separate characters:

```
>>> newlist2 = list("aatt")
>>> newlist2
['a', 'a', 't', 't']
>>> newlist2 = list(seq)
>>> newlist2
['a', 'g', 'g', 'N', 'a', 't', 't', 'g', 'N', 'a', 'g', 'g', 'g', 't',
'g', 'N', 'a', ’t']
```

Other very useful list functions include:
append—add an object to the end of a list
insert—add an object at the specified position within a list
pop—remove and show an element of the list
sort—order a list (in place)
reverse—reverse the elements in a list
join—attach the elements in a list to each other, separated by the chosen delimiter

```
>>> newlist
['agg', 'attg', 'agggtg', 'at']
>>> newlist.append("t")
>>> newlist
['agg', 'attg', 'agggtg', 'at', 't']
>>> newlist.insert(2, "GG")
>>> newlist
['agg', 'attg', 'GG', 'agggtg', 'at', 't']
>>> newlist.pop(4)
'at'
>>> newlist
['agg', 'attg', 'GG', 'agggtg', 't']
```

```
>>> newlist.sort()
>>> newlist
['GG', 'agg', 'agggtg', 'attg', 't']
>>> newlist.reverse()
>>> newlist
['t', 'attg', 'agggtg', 'agg', 'GG']
>>> newstring = "".join(newlist)
>>> newstring
'tattgagggtgaggGG'
>>> newstring = " ".join(newlist)
>>> newstring
't attg agggtg agg GG'
```

Exercise
Write a set of commands to print the reverse (not the reverse complement) of the sequence 'ACTCGAACGTGTGTCGTTCGGGATTACG'.

Tips:
 Assign the sequence to a variable
 Create a list with the characters of the sequence
 Reverse the list
 Concatenate the elements of the list together to form a string

Tuple
A tuple is created using the ( ) operator, and is a list that cannot be modified (immutable). The only commands that can be used on a tuple are 'count' and 'index.'

```
>>> newtuple = ("tt", "aa", 1)
>>> newtuple
('tt', 'aa', 1)
>>> newtuple[2]
1
>>> newtuple[2] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Dictionary
The dictionary is probably the most useful object in Python! it's an associative array, with every value assigned to a key. The key is a tuple and cannot be changed, but the values can be altered.

```
>>> codons = {'ATT':'I', 'CTT':'L', 'CCT':'P', 'ATC':'I'}
>>> print codons
{'CTT': 'L', 'CCT': 'P', 'ATT': 'I', 'ATC': 'I'}
>>> codons.keys()
['CTT', 'CCT', 'ATT', 'ATC']
```

```
>>> codons.values()
['L', 'P', 'I', 'I']
```

add a new element to the dictionary

```
>>> codons['AAA']='K'
>>> print codons
{'CTT': 'L', 'CCT': 'P', 'ATT': 'I', 'AAA': 'K', 'ATC': 'I'}
```

and delete one

```
>>> del codons['CTT']
>>> print codons
{'CCT': 'P', 'ATT': 'I', 'AAA': 'K', 'ATC': ‘I'}
```

You’ll get an error if you try to access a key that doesn’t exist:

```
>>> codons['GGG']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: ‘GGG’
```

There are a couple of ways you can use to check whether a key is present in a dictionary

```
>>> 'CCT' in codons.keys()
True
>>> codons.has_key('ATC')
True
```

Flow control

Python syntax is organized by indentation, not parentheses as in most other languages. All statements within a block of syntax must be indented the same way, though there is no required indentation style (four spaces is most commonly used).

basic styles:
if…elif…else
while
for
other commands such as pass, continue, break

if…elif…else

for two conditions:

```
>>> n = 12
>>> if n < 11:
...     print "less than 11"
... else:
...     print "greater than 11"
...
greater than 11
```

for multiple conditions, use the elif statement (just be careful about your conditions so that they don’t overlap in unexpected ways):

```
>>> GC = 0.2
>>> if GC < 0.4:
...     print "low GC content"
... elif GC > 0.7:
...     print "high GC content"
... else:
...     print "average GC content"
...
low GC content

>>> GC = 0.6
>>> if GC < 0.4:
...     print "low GC content"
... elif GC >= 0.4 and GC <= 0.7:
...     print "average GC content"
... else:
...     print "high GC content"
...
average GC content
```

while

let’s count to 10:

```
>>> n = 0
>>> i = 1
>>> n = n+1
>>> n
2
>>> n = n+1
>>> n
3
```

This is boring. Any time you do something that is repetitive and boring, you should probably be writing a program to do it more reproducibly. The `while` loop will execute a command until a particular condition is met. Be sure that the condition will eventually be met!

```
>>> n=0
>>> while n<= 10:
...    n=n+1
...    print n
...
1
2
3
4
5
6
```

```
7
8
9
10
11
```

interesting. Why does it go to 11?

Note: a shorthand for `n = n+1` is `n+=1` and you will see this more often. Works with other numbers and operations too (`n+=-2, n-=1, n/=5`)

Exercise
Fix the above program so that it only counts to 10. There are two ways to accomplish this.

Try something more complicated—let's add up all numbers from 1 to 100 (famous problem).

```
>>> n=0
>>> i=1
>>> while i<= 100:
...     n=n+i
...     i=i+1
...
>>> n
5050
>>> i
101
```

What happens if you leave out i=i+1 in this code? Why? (you can kill this with control-C)

for
The for statement is used for sequential looping, executing the code for each value in a sequence, list, dictionary etc.

The range operator is handy and is used often in for loops:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> for i in range(5):
...     print i
...
0
1
2
3
4
```

You can loop over a string:

```
>>> mystring = "string"
>>> for mycharacter in mystring:
...     print mycharacter
```

```
...
s
t
r
i
n
g
```

here I’m assigning each object in mystring to the variable mycharacter, and printing that variable each time.

You can loop over a list:

```
>>> newlist = ['t', 'attg', 'agggtg', 'agg', 'GG']
>>> for item in newlist:
...     print item
...
t
attg
agggtg
agg
GG
```

You can also loop over a dictionary (very common application):

```
>>> codons.items()
[('CCT', 'P'), ('ATT', 'I'), ('AAA', 'K'), ('ATC', 'I')]
>>> for key, value in codons.items():
...     print key, "=", value
...
CCT = P
ATT = I
AAA = K
ATC = I
```

Exercise
Rewrite the while loop that sums all numbers from 1 to 100, as a for loop.

Exercise
Write a set of commands that print the reverse complement of the sequence ‘ACTCGAACGTGTGTCGTTCGGGATTACG'
Tip:
  create a dictionary whose key:value pairs will give you complementary nucleotides
  follow the steps from the last exercise and use that dictionary, with a for or while loop, to find the complementary nucleotide for each base in the list you’ve created, before concatenating.

Exercise
If you used a for loop in the above exercise, rewrite it using a while loop. If you used a while loop, now write a for loop.

pass, continue, and break are also important in control flow in both for and while loops:

```
>>> n=-5
>>> if n > 100:
...    print "large"
... elif n > 10:
...    print "medium"
... elif n > 0:
...    print "small"
... else:
...    pass
...

>>> for i in range(10):
...     if i%2==0:
...         continue
...     elif i==5:
...         break
...     print i
...
1
3
```

Functions

Write a set of commands that removes all non-acgt letters from the string, “ccF $334 GGtt”.
Tips:
  make sure the string is either all uppercase or all lowercase
  create a dna string, ‘actg’ or ‘ACGT’ (depending on whether you’re using a lowercase or uppercase string) that you’ll match to
  create a new empty string
  check your input string, character by character, and if it matches your dna string, then append that character to the empty string (you can use the += operator to do this).

Now we have a useful DNA cleaning routine. If we want to do this often, it will be tedious to rewrite these steps each time. Fortunately, we don’t have to, because we have functions.

```
>>> def cleanDNA(sequence):
...     '''removes non-ACTG characters from an input string'''
...     dna_alphabet = 'actg'
...     outputdna = ''
...     for c in sequence.lower():
...         if c in dna_alphabet:
...             outputdna += c
...     return outputdna
...
>>> cleanDNA("ccF  $334 GGtt")
‘ccggtt'
```

In practice you will want to extensively document your functions, including usage. This is the absolute minimal documentation.

```
>>> cleanDNA.func_doc
'removes non-ACTG characters from an input string’
```

Exercise
Write a function that cleans a string as above, and returns the reverse complement of the cleaned DNA string.

Extra task:
Add a counter, so that the function reports the length of the longest run of consecutive non-DNA characters.