

Python 101

Control flow

FRANCISCO PINA-MARTINS

Day Overview:

Today we will go around the basics of control flow:

- What types are there?
- What does it do?
- How do I work with it?

Types:

There are essentially 2 types of flow control:

- Conditionals
- Loops

Conditionals:

"Boolean Operators"

Let's take a short trip back to the land of *High School Mathematics*:

```
<  » Less than;  
<= » Less or equal than;  
>  » Greater than;  
>= » Greater or equal than;  
!=  » Not equal to;  
<> » Not equal to (alternative);  
==  » Equal to (Note the double "=");
```

"Boolean expressions"

```
x is y  
x is not y  
x in y  
x not in y
```

Conditional Statements:

When are they used?

When we want our program to do different things if (http://docs.python.org/reference/compound_stmts.html#if) a determined condition is met.

How do they work?

Let's look at some pseudo-code:

```
if «Condition(s) to be met»:
    «Do something»
    «Do something else»
elif «Another condition»:
    «Do something different»
else:
    «Do something else entirely»
«Do something every time,»
«Since this part is not indented»
```

Take special care with:

- You can have as many "elif"s as you wish;
- You can only have one "else" and it has to be after the last "elif";

Real code example:

Let's look at a real code example:

```
1 sequence = "ATG"
2 if sequence == "ATG":
3     print "We have a start codon!"
4 elif sequence in ["TGA", "TAG", "TAA"]:
5     print "We have a stop codon!"
6 else:
7     print "Our sequence is neither a start nor a stop
8     codon."
```

We have a start codon!

- Try changing the value of *sequence* and see the different results.

Likewise, we can use other boolean operators:

```
1 sequence = "ATG"
2 if len(sequence) == 3:
3     print "This can be a codon."
4 elif len(sequence) > 3:
5     print "This is too long for a codon."
6 else:
7     print "This is too short for a codon."
8
```

```
This can be a codon.
```

- Notice the use of the `len()` (<http://docs.python.org/library/functions.html#len>) function. It is used to return the length of an object, in this case, the length of the string *sequence*.

The for loop:

When are they used?

When we want our program to do the same thing to a lot of things. The `for` loop (http://docs.python.org/reference/compound_stmts.html#for) will do something **for** every value in an `iterable` (<http://docs.python.org/glossary.html#term-iterable>).

How do they work?

Let's look at another pseudo-code example:

```
for «item» in «iterable»:
    «Do something with item»
    «Do something else with item»
«Do something after the loop is done»
```

Take special care with:

- An *iterable* can be any iterable object, such as:
 - A string, a tuple a list or a dictionary;
 - Characters in a string;
 - Elements of lists and tuples;
 - Keys and values of dictionaries;
 - Integers and floats are **not** iterable;
 - A *list* of integers, however is iterable;

Real code example:

Once again, let's look at a real code example:

```
1  for numbers in range(4):
2      print(numbers)
3
```

```
0
1
2
3
```

- Running this code will print the numbers from 0 to 3 (remember python starts to count from 0), each followed by a newline character.
- Also make note of the `range()` (<http://docs.python.org/library/functions.html#range>) function. It is used in this case to create a list of integers from 0 to 5 on the fly. It is a very versatile function, you can read more about it in the documentation.
- Another example could be:

```
1  sequences = ["ATGCTAGCTGATC", "ATGCCCTGATTAT"]
2  for i in sequences:
3      print(i)
4
```

```
ATGCTAGCTGATC
ATGCCCTGATTAT
```

Now that was easy, wasn't it? Let's make it a bit more difficult...

Nested loops:

Sometimes we have some code that we want to run **x** times and some code within that code that we want to run **y** times.

- In this example we want to find which sequences are common to both lists:

```
1 sequences1=["ATGTCTA", "TCGATCGA", "GCCCTAGT"]
2 sequences2=["ATCGCTA", "GCTATATT", "TCGATCGA"]
3 for i in sequences1:
4     for j in sequences2:
5         if i == j:
6             print "Sequence %s is common to both
7 lists" %(j)
```

Sequence TCGATCGA is common to both lists

Take special care with:

- Nested loops can look like a good idea at first, but they usually have a great impact on performance. If you are working with large datasets, you are advised to avoid them.

The while loop:

When are they used?

The [while](http://docs.python.org/reference/compound_stmts.html#while) (http://docs.python.org/reference/compound_stmts.html#while) loop is used when we want to combine the functions of the *if* statement and the *for* loop (sort of).

How do they work?

Here is some more pseudo-code as an example:

```
while «Condition is true»:
    «Do something»
    «Do something else»
«Do something after Condition is not true»
```

Take special care with:

- Make sure the contents of your *while* loop alter the condition being verified, otherwise you may get caught in an "infinite loop".

Real code example:

Let's look at another real code example:

```
1 number=0
2 while number <= 3:
3     print number
4     number += 1
5
```

```
0
1
2
3
```

Running this code will yield the same result as our first *for* loop, but it's done in a different way.

As you can see, the *while* loop will test against a condition and run the code in it while the condition is true.

Here's another example (a bit more bio and a bit less abstract). Let's call it an ORF generator:

```
1 import random
2 ORF = "ATG"
3 bases = ["A", "T", "G", "C"]
4 stops = ("TGA", "TAG", "TAA")
5 while ORF.endswith(stops) == False:
6     ORF += random.choice(bases)
7 print ORF
8
```

```
ImportError: No module named random
```

Wow, wait a minute, what is this? Let's look at it in parts. (Next slide please!)

The Mighty ORF generator:

```
import random
```

This will import the functions from the *random* module. Don't worry about it for now. We will have more fun with modules later.

Then, we declare our variables: *ORF*, *bases* and *stops*, so far so

good.

Finally the loop:

```
while ORF.endswith(stops) == False:
```

What this means - "While the variable ORF does **not** end with any of the content of *stops* do this:"

```
ORF += random.choice(bases)
```

What this means - "**Add** a random character from *bases* to ORF."

Here is the documentation for the used functions: [endswith\(\)](http://docs.python.org/library/stdtypes.html#str.endswith) (<http://docs.python.org/library/stdtypes.html#str.endswith>), [random.choice\(\)](http://docs.python.org/library/random.html#random.choice) (<http://docs.python.org/library/random.html#random.choice>).

Can you see something wrong with this?

Deeper into control flow:

Break, continue, else on loops and pass:

- Break
 - will immediately stop any *for* or *while* loop
- Continue
 - will immediately continue with the next iteration of the loop
- Else on loops
 - will do something *after* the loop is finished
- Pass
 - will do absolutely nothing

Real code examples:

(We don't really need pseudo-code for this)

```
1 breakpoint = 4
2 skippoint = 2
3 for i in range(1,6):
4     if i == skippoint:
5         continue
```

```
6         if i == breakpoint:
7             print("loop broke at " + str(breakpoint))
8             break
9         print i
10    else:
11        print "loop never reached %s and never broke"
12    %(breakpoint)
```

```
1
3
loop broke at 4
```

Take special care with:

- The way the `range()` (<http://docs.python.org/library/functions.html#range>) function was used; In this case we also defined the *start* of the count;
- The `str()` (<http://docs.python.org/library/functions.html#str>) function - it will convert any object (in this case an *integer*) into a *string*. This is required to concatenate the variables in the `print()` (<http://docs.python.org/library/functions.html#print>) function;
- Try to change the **breakpoint** and **skippoint** variables for different results;

Special type of iteration - dictionaries:

- When "looping" through a dictionary, we can use a special function - `items()` (<http://docs.python.org/library/stdtypes.html#dict.items>)

```
1    d = {"one": "1", "two": "2", "three": "3"}
2    for key, value in d.items():
3        print key + " - " + value
4
```

```
one - 1
two - 2
three - 3
```

What's so special about this?

Note that we are *iterating* two variables at the same time. This can be tricky to master at first, but it is a very useful function once you've gotten the hang of it.

Take special care with:

- Dictionaries will not preserve the order that the *key:value* pairs are stored in;
 - This means that when you iterate through a dictionary, your *key:value* pairs can turn up in any order;
- You can do something similar with two (or more) lists by using the `zip()` (<http://docs.python.org/library/functions.html#zip>) function;

Biological examples:

Let's suppose we have a dictionary of 3 lists with several species each and we wish to know in which of these lists (if at all) we can find our species - *Homo sapiens*

```
1 listset = {"reptiles":["Lacerta lepida", "Psammodromus
2 algirus",
3 "Aspidoscelis ironata"], "plants":["Arabidopsis
4 thaliana", "Quercus suber",
5 "Vitis vinifera", "Ricinus comunis"], "mammals":["Mus
6 musculus",
7 "Canis lupus", "Homo sapiens"]}
8 species = "Homo sapiens"
9 for lists in listset:
10     if species in listset[lists]:
11         print(species + " can be found in the
12 following list: " + lists)
13         break
14     else:
15         print(species + " could not be found in any of the
16 lists.")
```

Homo sapiens can be found in the following list: mammals

Take special care with:

- Notice that when defining *listset*, the code is split along several lines; you can read more about this [here \(http://www.python.org/dev/peps/pep-0008/#maximum-line-length\)](http://www.python.org/dev/peps/pep-0008/#maximum-line-length);
- In line 7, we are calling the values in the dictionary, not the keys;

- Try changing the variable *species* and see the results;

Biological examples (part II):

In this example we have a string with 3 "columns" divided by tabs ("\t") in python. Let's suppose that we wish to extract the Fst value for each column into a list.

```
1  datastring = ""# Locus ID    Overall Pi  Fst
2  2      0.4 0.1666666667
3  3      0.5 0.0000000000
4  4      0.1 0.1095890411
5  5      0.2 0.2068965517""
6  datalist = datastring.splitlines()
7  fsts = []
8  for lines in datalist:
9      if lines.startswith("#"):
10         pass
11     else:
12         values = lines.split("\t")
13         fst = values[2]
14         fsts.append(fst)
15 print(fsts)
17
```

```
['0.1666666667', '0.0000000000', '0.1095890411',
'0.2068965517']
```

Take special care with:

- The `splitlines()` (docs.python.org/library/stdtypes.html#str.endswith) type; this built-in will split a string into a list where each element is a line of the string;
- The `startswith()` (<http://docs.python.org/library/stdtypes.html#str.startswith>) function; it is pretty much self explanatory;
- The `split()` (<http://docs.python.org/library/stdtypes.html#str.split>) function; it will split a string into a list of words eliminating the separator.

- You have to test this in IDLE or equivalent.
-