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 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 enteirely»
«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:
```

```
    print "This can be a codon."
elif len(sequence) > 3:
    print "This is too long for a codon."
else:
    print "This is too short for a codon."
```

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

- Notice the use of the 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 will do something **for** every value in an iterable.

How do they work?

Let's look at another pseudo-code example:

```
for «item» in «iterable»:
    «Do somthing 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:

```
for numbers in range(4):
    print(numbers)
```

```
0
1
2
3
```

- Running this code will print the numbers from 0 to 3 (remeber python starts to count from 0), each followed by a newline character.
- Also make note of the 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:

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

```
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:

```
sequences1=["ATGTCTA", "TCGATCGA", "GCCCTAGT"]
sequences2=["ATCGCTA", "GCTATATT", "TCGATCGA"]
for i in sequences1:
    for j in sequences2:
        if i == j:
            print "Sequence %s is common to both 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 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 diffrent 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
9
```

```
ATGGGATCGAGGTTACTGA
```

Wow, wait a minuite, 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(), 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 and only* the loop is finished. Breaking the loop will not run this code;
- Pass
  - will do absolutely nothing;

Real code examples:

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

```
breakpoint = 4
skippoint = 2
for i in range(1,6):
    if i == skippoint:
        continue
    elif i == breakpoint:
        print("loop broke at " + str(breakpoint))
        break
    print i
else:
    print "loop never reached %s and never broke"
%(breakpoint)
```

```
3
loop broke at 4
```

Take special care with:

- The way the range() function was used; In this case we also defined the *start* of the count;
- The 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() 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()

```
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() 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
    algirus",
```

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

```
            fsts.append(fst)
    print(fsts)
```

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

Take special care with:

- The splitlines() method; this built-in will split a string into a list where each element is a line of the string;
- The startswith() method; it is pretty much self explanatory;
- The split() method; it will split a string into a list of words eliminating the separator.
- You have to test this in IDLE or equivalent.