EEB 177 Lecture 10

Thursday Feb 10th, 2017

Topics

- for, if, else, while statements
- functions
- reading and writing files

Office Hours

Today 1045-1145 **Wednesday 10-11**

for loops have the following structure:

```
for x in y:
    do something
```

y is a list (or list-like object) x is a variable names the colon sets off an indented block of code (the body of the loop)

challenge

1. write a for loop that prints each letter of your partner's name.
2. write a for loop prints that takes the following dictionary and prints out a statement indicating whether the value of each key is an even or odd number. {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

the range function

```
for number in range(6):
    print(number)
```

more on range

With two numbers, range will count up from the 1st number (inclusive) to the second (exclusive):

```
for number in range(3, 8):
    print(number)
```

range and step

With three numbers, range will count up from the 1st to the second with the step size given by the third:

```
for number in range(2, 14, 4):
    print(number)
```

while loops

a while loop runs until some condition is met.

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

program flow in python

In its simplest form, a program is just a series of instructions (statements) that the computer executes one after the other. In Python, each statement occupies one line (i.e., it is terminated by a newline character). Other programming languages use special characters to terminate statements (e.g., ; is used in C).

Lets demonstrate statements that control flow in python with a simple program.

conditional tests

A condition is simply a bit of code that can produce a true or false answer.

```
print(3 == 5)
print(3 > 5)
print(3 <=5)
print(len("ATGC") > 5)
print("GAATTC".count("T") > 1)
print("ATGCTT".startswith("ATG"))
print("ATGCTT".endswith("TTT"))
print("ATGCTT".isupper())
print("ATGCTT".islower())
print("V" in ["V", "W", "L"])
```

we use conditional tests to control the flow of our program

```
expression_level = 125
if expression_level > 100:
    print("gene is highly expressed")
```

if, elif, else

if and **else** create branching points in your program resulting in the execution of different blocks of code depending on a condition.

```
x=4
if x % 2 == 0:
    print("Divisible by 2") #body of loop
```

note indentation to designate loop body

We can use **else** to specify an action when a condition is not met.

```
x=4
if x % 2 == 0:
    print("Divisible by 2")
else:
    print("Not divisible by 2")
```

```
expression_level = 125
if expression_level > 100:
   print("gene is highly expressed")
else:
    print("gene is lowly expressed")
```

If we have multiple cases that need to be evaluated, **elif** is useful...

```
x = 17
if x % 2 == 0:
    print("Divisible by 2")
elif x % 3 == 0:
    print("Divisible by 3")
elif x % 5 == 0:
    print("Divisible by 5")
elif x % 7 == 0:
    print("Divisible by 7")
else:
    print("Not divisible by 2, 3, 5, 7")
```

Challenge

change the program below so that it reports the temperature in Fahrenheit or Celsius depending on user input.

```
xx = input("What is the temperature in Fahrenheit?")
yy = (float(xx) - 32) * 5 / 9
print("The temperature in Celsius is {}".format(yy))
```

if, elif, and else part II

we can handle complex flow with these statements.

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    else:
        file2.write(accession + "\n")
```

use **and**, **or** and **not** to specify complex conditionals

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a') and accession.endswith('3
        print(accession)
```

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a') or accession.startswith('b
        print(accession)
```

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for acc in accs:
    if acc.startswith('a') and not acc.endswith('6'):
        print(acc)
```

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
file3 = open("three.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    elif accession.startswith('b'):
        file2.write(accession + "\n")
    else:
        file3.write(accession + "\n")
```

Reading and writing files

To work with text file contents you need to first open the file and then read the contents in as a string

```
# open the file
my_file = open("dna.txt")
# read the contents
my_dna = my_file.read()
# calculate the length
dna_length = len(my_dna)
# print the output
print("sequence is " + my_dna +  " and length is " + str(dn
```

Functions

Functions are blocks of code that do something useful. You have already used many built-in functions of strings (**str.lower()**, **file.write()**).

Functions take **arguments**. You define functions in python with the **def** keyword.

```
def stringAnalyzer(ss):
    length = len(ss)
    print("This string has {} characters".format(length))

tt = stringAnalyzer("hello")
This string has 5 characters
```

This function expects a string, counts the characters in it, and prints the total length. No value is returned in this case.

Returning values

More commonly your function will do something to the arguments you provide and return the modified values.

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

Variable scope

Variables created within functions cannot be used outside of that function.

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

Improving functions

One advantage to functions is that they allow you edit and improve your code in a modular way. Here we will modify our code to avoid long decimals and an incorrect AT content when lowercase values are passed to the function **For the these examples we want our function to work correctly on the following four lines.**

```
#we want our function to properly execute these lines
my_at_content = get_at_content("ATGCGCGATCGATCGAATCG")
print(str(my_at_content))
print(get_at_content("ATGCATGCAACTGTAGC"))
print(get_at_content("aactgtagctagctagcagcgta"))
```

unimproved version

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

this gives some problems

```
 print(get_at_content("ATGCATGCAACTGTAGC"))
0.529411764706
 print(get_at_content("aactgtagctagctagcagcgta"))
0.0
```

Improved version

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, 2)

my_at_content = get_at_content("ATGCGCGATCGATCGAATCG")
print(str(my_at_content))
print(get_at_content("ATGCATGCAACTGTAGC"))
print(get_at_content("aactgtagctagctagcagcgta"))
```

Better! Here we have added upper() and round() methods to format values before they are returned. What if we wanted the number of decimal places to be an argument in our function so that the user could specify the decimal length?

Modify your function to do this Does it work?

Here is one way to control the number of significant figures

```
def get_at_content(dna, sig_figs):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)
```

Encapsulation

OK you have made changes to your function to improve formatting. What you may not have noticed is that you did not need to modify (much) the rest of the program outside the function. This demonstrates the idea of **encapsulation** which really just mean breaking up a big program in to smaller parts that can be managed in a straightforward way.
Keep this idea in mind as you start to consider you final projects!

Keyword arguments

If you define a function in the typical way, the order that you pass the arguments is critical.

```
In [62]: def outOfOrder(number, pet):
    ....:         print "I own {} {}s".format(number, pet)
    ....:
In [63]: outOfOrder(3, "dog")
I own 3 dogs
In [64]: outOfOrder("dog", 3)
I own dog 3s
```

Keyword arguments, wherein the name of the argument followed by an "=" followed by the value, can help avoid these kinds of errors.

```
outOfOrder(number=3, pet="dog")
#same as
outOfOrder(pet="dog", number=3)
```

Default values

It is often useful to pass a default value to a function that can be overridden.

```
def get_at_content(dna, sig_figs=2):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)

get_at_content("ATCGTGACTCG")
get_at_content("ATCGTGACTCG", 3)
get_at_content("ATCGTGACTCG", sig_figs=4)
```

Now the value will be written to two decimal places unless otherwise specified.

Testing functions

It is important that your code is behaving the way you expect it will. One way to do this is to invoke functions with arguments that should produce a known value. You can use the **assert** function to test if your code passes this test.

```
assert get_at_content("ATGC") == 0.5 #this should work
assert get_at_content("ATGCNNNNNNNNNN") == 0.5 #what about
```

Modify your get_at_content() program so that you function will pass an assert with "Ns"

Improved code

```
def get_at_content(dna, sig_figs=2):
    dna = dna.replace('N', '')
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)
```

Now the function passes the assert test.

It is a good idea to include a series of assert tests for functions you creates. You can easily comment the assert statements out as needed.

```
assert get_at_content("A") == 1
assert get_at_content("G") == 0
assert get_at_content("ATGC") == 0.5
assert get_at_content("AGG") == 0.33
assert get_at_content("AGG", 1) == 0.3
assert get_at_content("AGG", 5) == 0.33333
```

Debugging

the implementation of the python debugger (**pdb**)Ipython provides another way of searching out bugs. Lets examine it. Start with this function.

```
def createabug(x):
    y = x**4
    z = 0.
    y = y/z
    return y

createabug(5)
```

What happens?

Use your text editor to ave this file as *debugme.py* in your classwork directory, Load and execute your function:

```
In [1]: %run debugme.py
...
----> 4      y = y/z
      5      return y
      6

ZeroDivisionError: float division by zero #doesn't work...
```

Now start the debugger in ipython and run again:

```
In [72]: %pdb
Automatic pdb calling has been turned ON
In [73]: %run debugme.py
...
ZeroDivisionError: float division by zero
> /home/vagrant/scripts/debugme.py(4)createabug()
      3     z = 0.
----> 4     y = y/z
      5     return y

ipdb> # <---- NOTICE THIS!
```

Now we are in the debugger shell!

pdb commands

Within the debugger we can move around in our code and examine variables to see what is happening.

- **n** move to the next line.
- **ENTER** repeat the previous command.
- **s** "step" into function or procedure (i.e., continue the debugging inside the function, as opposed to simply run it). p x print variable x.
- **c** continue until next break-point.
- **q** quit
- **l** print the code surrounding the current position. r continue until the end of the function.

try out pdb

```
ipdb> p x
25
ipdb> p y
390625
ipdb> pz
*** NameError: name 'pz' is not defined
ipdb> px
*** NameError: name 'px' is not defined
ipdb> p x
25
ipdb> p y
390625
ipdb> p z
0.0
ipdb> p y/z
*** ZeroDivisionError: ZeroDivisionError('float division by
ipdb> q
In [74]: pdb
Automatic pdb calling has been turned OFF
```

Automatically launch pdb

If you want to start pdb within a longer script, place this line at the point you want to examine:

```
import pdb; pdb.set_trace()
```