Iteration

Iteration is a common feature of computer programs. Iteration is the repetition of a specific set of statements a specified number of times or while some condition is true. Python has several features that make iteration easy.

In [3]:

```
def sum_num(n1, n2, n3 = 0, n4 = 0):
    return n1+n2+n3+n4
sum_num()
```

Out[3]:

```
0
```

`while` loops

`while` loop statements are used to repeat blocks of code while some condition is true:

```
while some condition is True:
  do something
```

For example:

In [4]:

```
countdown(n):
while n > 0:
    print(n)
    n -= 1 # subtract 1 from n, equivelent to n = n - 1, this is called c
ntdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
```

In plane English the code reads:

"While n is greater than 0, print n, then subtract 1 from n."

So while n is greater than 0, the `while` statement evaluates to true and the block of code is executed. After n iterations, n is reduced to 0, the `while` condition than evaluates to False, and the code is not executed.

Infinite loops

Within the body of the `while` loop the value of a variable is often changed such that the condition tested in the `while` statement no longer evaluates to `True` after some number of iterations and the loop terminates. Otherwise, and infinite loop is born.

Exercise 5a

Write a function within a script (`countup(n)`) that contains a `while` loop that counts up from n. Recall that to execute the code from within the script, you need an `if __name__ == '__main__':` block.

```
In [ ]:
```

I just tricked into you creating an infinite loop. To escape, remember to use `ctrl-c` .

`break` and `continue`

Although infinite loops are often unintentional, they can actually be quite useful. Write a function (`seq_len()`) that prompts the user for a sequence and prints the length of the sequence. The function should repeat the task until the user no longer provides a sequence and just hits return:

In [ ]:

```
def seq_len():
    while True:
        seq = input('Enter a sequence: ')
        if seq == None:
            print('done')
            break
        else:
            print(len(seq))
seq_len()
```

```
Enter a sequence: atg
3
Enter a sequence:
0
Enter a sequence:
0
Enter a sequence:
0
Enter a sequence:
0
```

The `while` statement will always evaluate to `True` because `True` is True. The `break` keyword provides a convenient way to exit the entire loop in which it's contained. Thus, you can evaluate whether or not a condition is `True` at any point within the loop and exit the loop if desired - an affirmative approach (stop when some condition is True) rather than a negative approach (keep going until some condition is False).

Let's modify the `seq_len()` function to ignore input that is shorter than 3 characters:

In [1]:

```
def seq_len():
    while True:
        seq = input('Enter a sequence: ')
        if seq  == '':
            print('done')
            break
        elif len(seq) < 3:
            continue
        else:
            print(len(seq))
        print('test')
seq_len()
```

```
Enter a sequence:
done
```

The keyword `continue` skips over any remaining code and goes back to the `while` statement for another iteration.

Let's write a function `inverse()` that calculates the inverse of a number. In code outside of the function we'll use a `while` loop to verify that a valid number is entered by the user:

In [2]:

```
def inverse(n):
    return 1/n

num = float(input('Enter a number: '))
while num == 0:
    num = float(input('Enter a valid number: '))
print(inverse(num))
```

```
Enter a number: 0
Enter a valid number: 0
Enter a valid number: 0
Enter a valid number: 5
0.2
```

Exercise 5b

Create a module (`stats.py`) with two functions:

1. `total()` reads numbers input by the user until they hit return instead of entering a number and then calculates the sum of the numbers input by the user.
2. `avg()` reads numbers input by the user until they hit return instead of entering a number and then calculates the average of the numbers input by the user.

Import the module into either jupyter notebook or the python shell and test each of the functions.

In [4]:

```
def total():
    s = 0
    n = input('Enter a number: ')
    while n!= '':
        s += float(n)
        n = input('Enter another number: ')
    return s

def avg():
    s = 0
    c = 0
    n = input('Enter a number: ')
    while n != '':
        s += float(n)
        c += 1
        n = input('Enter another number')
    return s/c

print(avg())
```

```
Enter a number: 5
Enter another number5
Enter another number4
Enter another number4
Enter another number
4.5
```

for loops

`for` loops are for looping over a defined list of objects. `for` loops are typically used when we want to repeat a block of code a fixed number times, as opposed to a `while` looop in which we want to repeat something until some condition is met:

```
for some elements in a sequence:
    do something
```

For example:

In [9]:

```
seq = 'ATGTATA'
for nt in seq:
    print(nt, end = '')
```

```
ATGTATA
```

In plane English, the code reads: "For each nucleotide in the variable seq, print the nt."

Let's write a function, `comp()`, that returns the complement of a sequence:

In [12]:

```
seq = 'ATGTATA'
def comp(s):
    c = ''
    for nt in seq:
        if nt == 'A':
            c += 'T'
        elif nt == 'T':
            c += 'A'
        elif nt == 'G':
            c += 'C'
        elif nt == 'C':
            c += 'G'
        else:
            print('Non DNA characters ignored')
    return c

print(comp(seq))
```

```
TACATAT
```

The `range` function

To iterate over a simple list of number, use the `range` function. The `range()` function allows you to specify a range of integers to iterate through using the following syntax: `range(start, stop[, step])`. Essentially, it generates a list of numbers between `start` and `stop` at optional `step` intervals which are generally iterated over in for loops.

We can use `help(function_name)` to get a description of a function:

In [14]:

```
help(range)
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (
inclusive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..
., j-1.
 |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1,
2, 3.
 |  These are exactly the valid indices for a list of 4 elements.
 |  When step is given, it specifies the increment (or decrement).
 |
 |
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate s
ignature.
 |
 |  __reduce__(...)
 |      helper for pickle
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(...)
 |      Return a reverse iterator.
 |
 |  count(...)
 |      rangeobject.count(value) -> integer -- return number of occurr
ences of value
 |
 |  index(...)
 |      rangeobject.index(value, [start, [stop]]) -> integer -- return
index of value.
 |      Raise ValueError if the value is not present.
 |
 |  ------------------------------------------------------------------
----
 |  Data descriptors defined here:
 |
 |  start
 |
 |  step
 |
 |  stop
```

Here's a simple example of `range()` in action:

In [15]:

```
for i in range(0,10,1):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Note that the `stop` number in the range is not part of the sequence. By default, if only one argument is given, it's treated as the `stop` value, `start` defaults to `0` and `step` defaults to 1:

In [16]:

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

If two arguments are passed to the function, `step` defaults to 1:

In [19]:

```
for i in range(1,10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

Let's print all even numbers between 1-10:

In [20]:

```
for i in range(2,11,2):
    print(i)
```

```
2
4
6
8
10
```

Print all odd numbers less than 9:

In [21]:

```
for i in range(1, 9, 2):
    print(i)
```

```
1
3
5
7
```

`range()` can also iterate up from a negative numbers but the the number with the lower value must be the `start` value:

In [22]:

```
for i in range(-10,0):
    print(i)
```

```
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
```

To iterate from a higher number to a lower number, use negative values for `step` . For example, print each number from 10 to 0:

In [25]:

```
for i in range(10, -1, -1):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
0
```

Let's write a function, `sum_int(n)` , that sums the integer numbers between 1 and n:

In [27]:

```
def sum_int(n):
    # sum numbers between 1 and n
    s = 0
    for i in range(1, n+1):
        s += i
    return s
sum_int(10)
```

Out[27]:

```
55
```

Write a function, `even_numbers(n)` , that sums all even numbers less than n:

In [29]:

```
def even_numbers(n):
    s = 0
    for i in range(0,n,2):
        s += i
    return s

even_numbers(10)
```

Out[29]:

```
20
```

Anything that can be done with a `for` loop, can also be done with a `while` loop, but not vice versa. For example, revise the `even_numbers()` function from above to use a `while` loop instead of a `for` loop:

In [30]:

```
def even_numbers(n):
    s = 0
    i = 0
    while i < n:
        s += i
        i += 2
    return s
even_numbers(10)
```

Out[30]:

```
20
```

Notice that the `while` loop required extra code, making it less compact, and also more difficult to read.

Additional Exercises

Use whatever method you prefer to test your code.

5c) Write a function, `reverse()` , that returns the reverse of a DNA or RNA sequence. Hint: you can concatenat something to either the end or the begining of a variable. For example:

```
x += z # end
x = x + z # also end, same as above
x = z + x # beginning
```

Use the empty code cell below to test this.

5d) Write a function, `nt_counter()` , that prompts the user for a DNA sequence and returns the number of As, Cs, Ts, and Gs it contains. Your function should contain an infinite loop such that it continues to prompt the user for a sequence and computes the numbers of each nucleotide until the user just hits return.

In [ ]:

Present | Slides | Themes | Help