

An Introduction to Python

Day 2

Renaud Dessalles

dessalles@ucla.edu



Data structures

Python's Data Structures - Lists

- * Lists can store lots of information.
- * The data doesn't have to all be the same type! (unlike many other programming languages)

```
>>> myList = []
>>> myEmptyList=[]
>>> myShoppingList=["Apples", "Beer", "Chocolate"]
>>> myShoppingList
['Apples', 'Beer', 'Chocolate']
>>> myMixedList = [123, "a string", 2.75]
>>> myMixedList
[123, 'a string', 2.75]
```

Python's Data Structures – Lists 2

- * Can access and change elements of a list by index.
- * Starting at 0
- * `myList[0]`
- * Just like strings.

```
>>> myNucList=['A','G','C','T']
>>> myNucList
['A', 'G', 'C', 'T']
>>> myNucList[2]
'C'
>>> myNucList[2]='G'
>>> myNucList[2]
'G'
>>> myNucList
['A', 'G', 'G', 'T']
```

Python's Data Structures – Lists 3

- * Lists have lots of handy functions.
- * `myList.function(arguments)`
- * Most are self explanatory.
- * Get an error if `index()` can't find what it's looking for.

```
>>> myNucList=['A','G','C','T']
>>> myNucList.append('C')
>>> myNucList
['A', 'G', 'C', 'T', 'C']
>>> len(myNucList)
5
>>> myNucList.index('G')
1
>>> myNucList.index('C')
2
>>> myNucList.index('B')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'B' is not in list
```

Python's Data Structures – Lists 4

```
>>> myNucList
['A', 'G', 'A', 'C', 'T', 'C']
>>> myNucList.sort()
>>> myNucList
['A', 'A', 'C', 'C', 'G', 'T']
>>> myNucList.remove('G')
>>> myNucList
['A', 'A', 'C', 'C', 'T']
>>> del myNucList[3]
>>> myNucList
['A', 'A', 'C', 'T']
>>> myNucList.pop(1)
'A'
>>> myNucList
['A', 'C', 'T']
>>>
```



3 ways to delete

Python's Data Structures – Lists 5

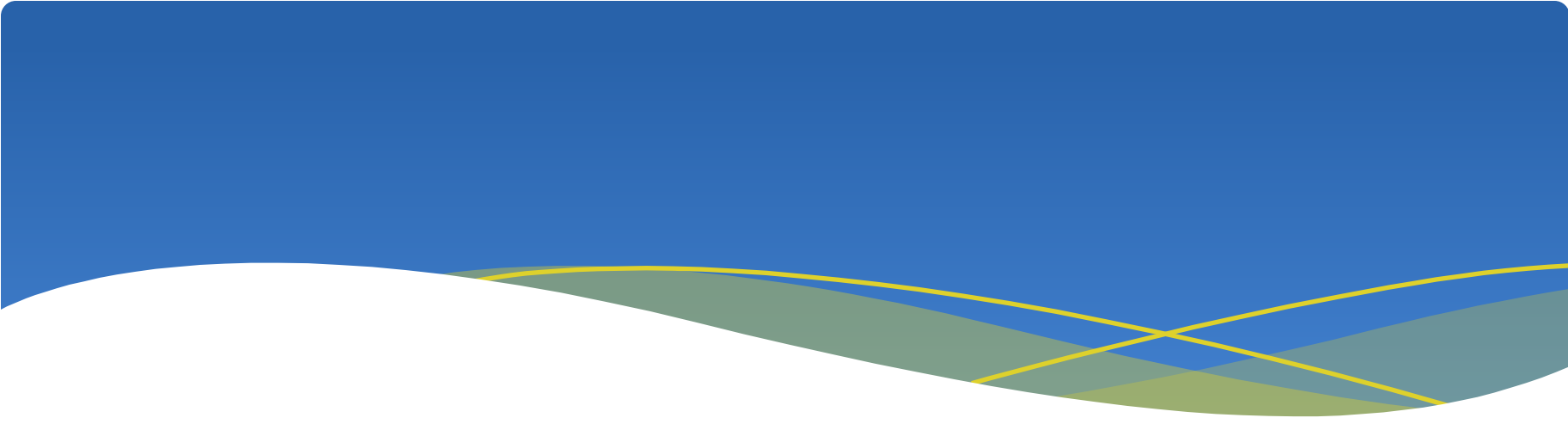
```
>>> letters = ['a','b','c','d','e','f']
>>> letters[0:2]
['a', 'b']
>>> letters[:4]
['a', 'b', 'c', 'd']
>>> letters[1:]
['b', 'c', 'd', 'e', 'f']
>>> letters[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> letters[6:7]
[]
```

Python's Data Structures – Dictionaries

- * Like lists, but have **keys** and **values** instead of index.
- * **Keys** are strings or numbers
- * **Values** are almost anything. E.g. Strings, lists, even another dictionary!

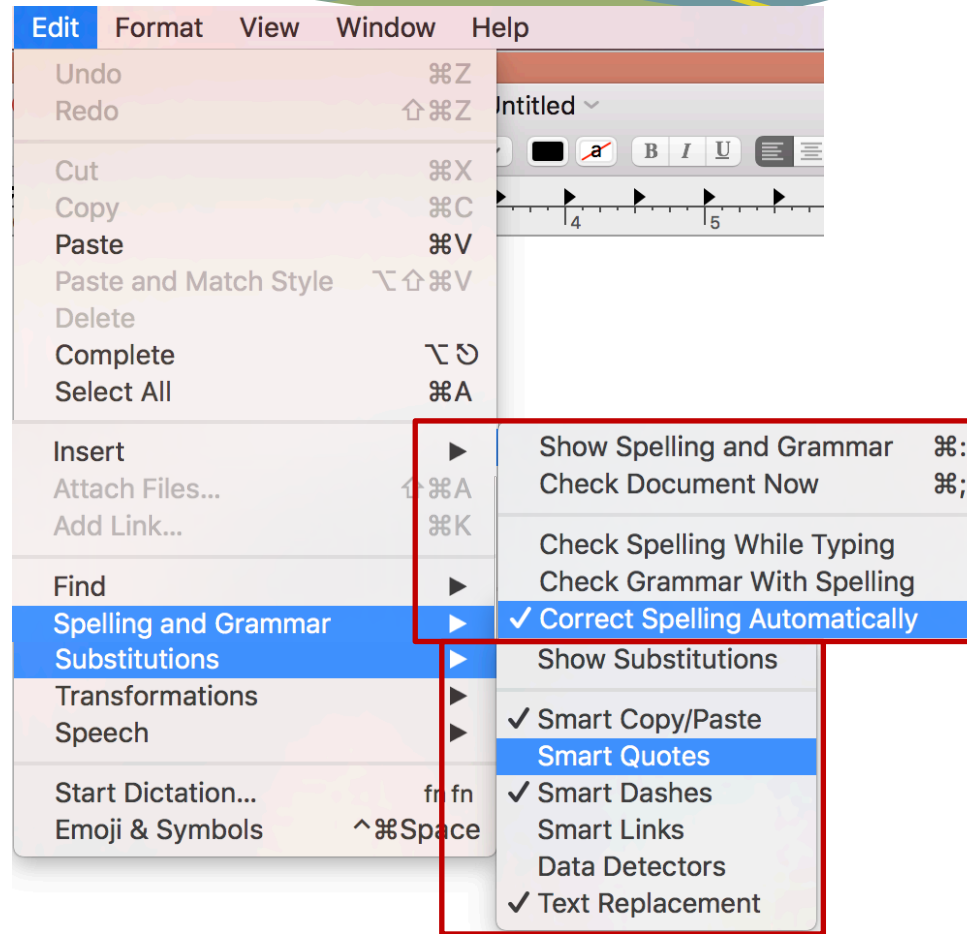
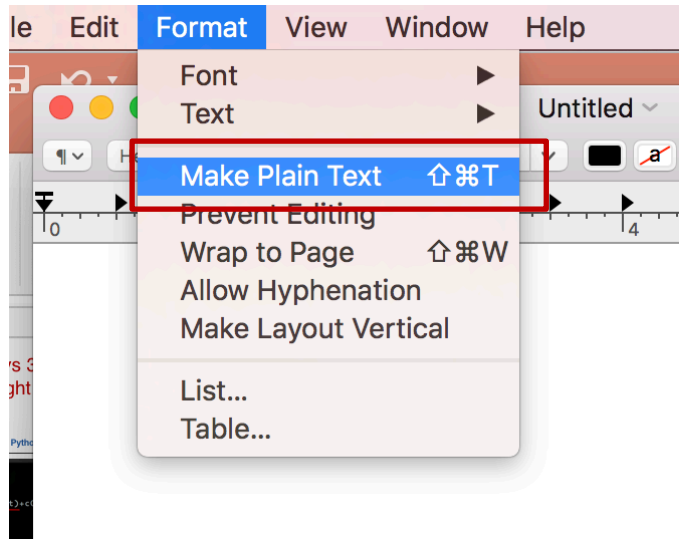
```
>>> experiments={'control':1000,'exp1':500,'exp2':600}
>>> experiments
{'control': 1000, 'exp2': 600, 'exp1': 500}
>>> experiments['exp2']
600
>>> experiments['control']
1000
>>> experiments['exp3']=300
>>> experiments
{'control': 1000, 'exp2': 600, 'exp1': 500, 'exp3': 300}
```

```
>>> del experiments['exp2']
>>> experiments
{'control': 1000, 'exp1': 500, 'exp3': 300}
```

Scripts

Reminder: writing in TextEdit (for Mac users)

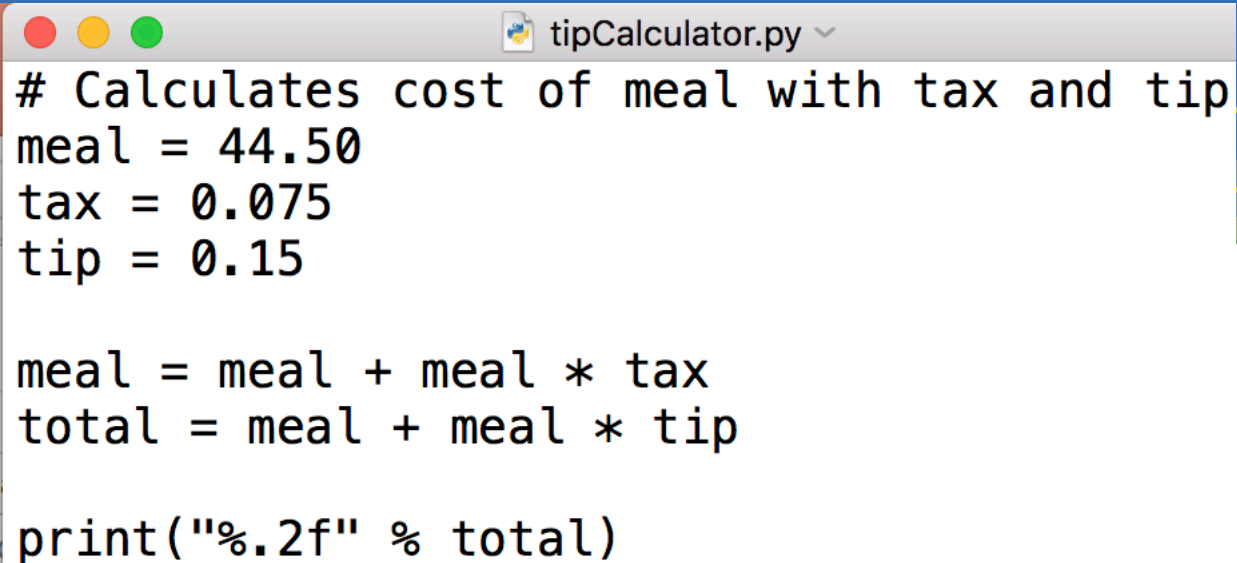


Example: Tip Calculator

- * First let's figure out the pseudocode:
 - * Set cost of meal
 - * Set rate of tax
 - * Set tip percentage
 - * Calculate meal + tax
 - * Calculate and return meal with tax + tip

```
tipCalculator.py ▼  
# Calculates cost of meal with tax and tip  
meal = 44.50  
tax = 0.075  
tip = 0.15  
  
meal = meal + meal * tax  
total = meal + meal * tip  
  
print("%.2f" % total)
```

A Script Not a Module



```
# Calculates cost of meal with tax and tip
meal = 44.50
tax = 0.075
tip = 0.15

meal = meal + meal * tax
total = meal + meal * tip

print("%.2f" % total)
```

```
[>>> exit()
QCBs-MacBook-Pro:~ qcbcollaboratory$ ls
Desktop                Movies                  tipCalculator.py
Documents              Music                  workshop.py
Downloads              Pictures
Library                Public
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 tipCalculator.py
55.01
```

Example: Tip Calculator 3

- * What if we want to calculate for a different meal cost without rewriting the code.
- * Pass the amount from the command line to python.

```
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 tipCalculator.py 55.00
```

- * How do we get python to understand the new amount?
- * Need to import sys
- * **sys.argv** is a list of strings of parameters passed from the command line.

Handling Commandline Arguments

```
tipCalculator.py
import sys

print(sys.argv)

# Calculates cost of meal with tax and tip
meal = 44.50
tax = 0.075
tip = 0.15

meal = meal + meal * tax
total = meal + meal * tip

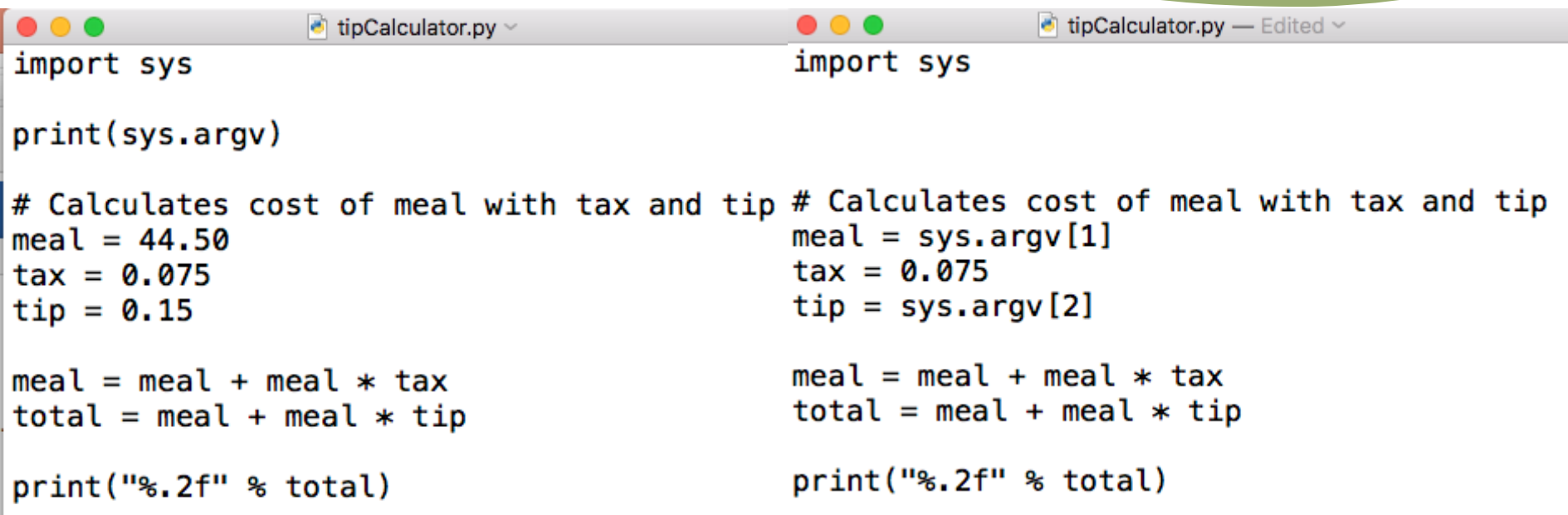
print("%.2f" % total)
```

```
IOCBs-MacBook-Pro:~ acbcollaboratory$ python3 tipCalculator.py arg1 arg2 arg3
['tipCalculator.py', 'arg1', 'arg2', 'arg3']
55.01
```

`sys.argv[0]` is `tipCalculator.py`
`sys.argv[1]` `arg1...`

Handling Commandline Arguments

2



```
import sys

print(sys.argv)

# Calculates cost of meal with tax and tip
meal = 44.50
tax = 0.075
tip = 0.15

meal = meal + meal * tax
total = meal + meal * tip

print("%.2f" % total)
```

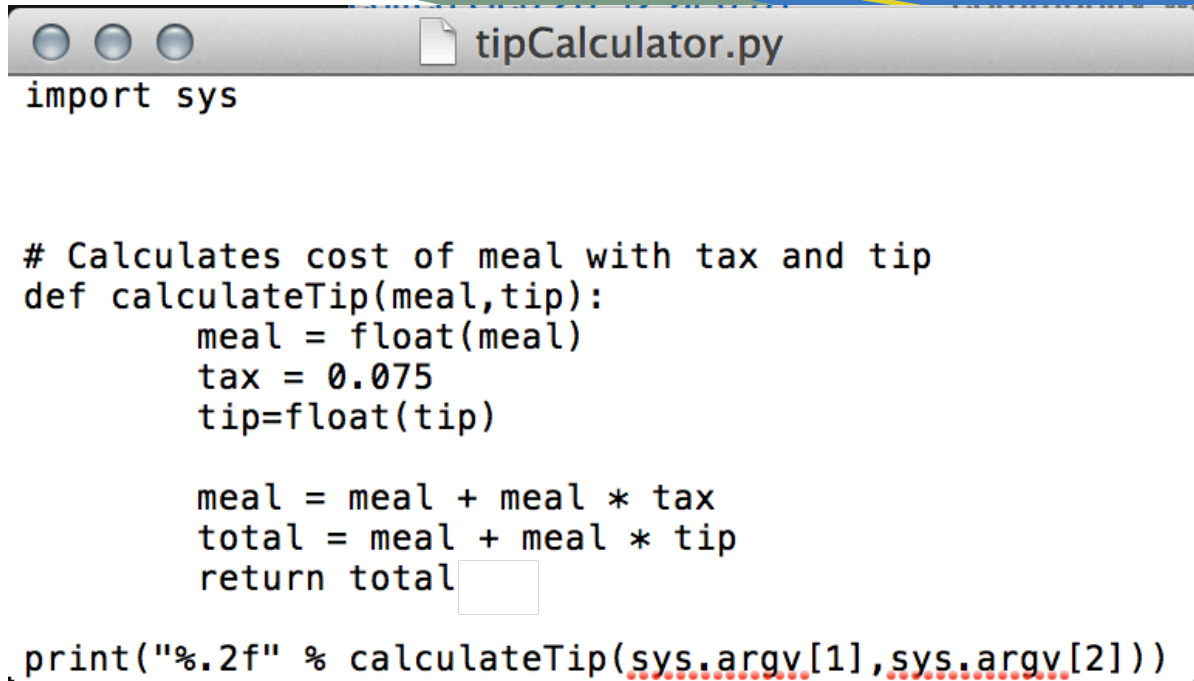
```
import sys

# Calculates cost of meal with tax and tip
meal = sys.argv[1]
tax = 0.075
tip = sys.argv[2]

meal = meal + meal * tax
total = meal + meal * tip

print("%.2f" % total)
```

Handling Commandline Arguments 3



```
import sys

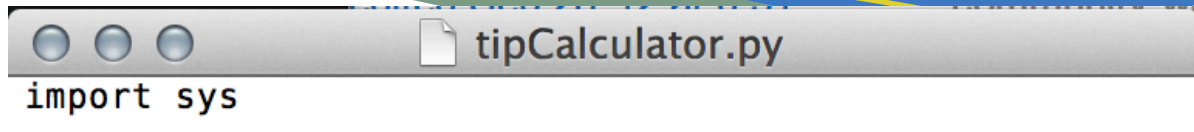
# Calculates cost of meal with tax and tip
def calculateTip(meal,tip):
    meal = float(meal)
    tax = 0.075
    tip=float(tip)

    meal = meal + meal * tax
    total = meal + meal * tip
    return total

print("%.2f" % calculateTip(sys.argv[1],sys.argv[2]))
```

- * Make calculateTip a function. Useful if we need to reuse that code in future programs!
- * Command line arguments from sys.argv are always strings so cast to a float if we want to do maths with them.

Test Your Tip Calculator



```
import sys

# Calculates cost of meal with tax and tip
def calculateTip(meal,tip):
    meal = float(meal)
    tax = 0.075
    tip=float(tip)

    meal = meal + meal * tax
    total = meal + meal * tip
    return total

print("%.2f" % calculateTip(sys.argv[1],sys.argv[2]))
```

```
[QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 tipCalculator.py 20.00 0.15
24.73
[QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 tipCalculator.py 20.00 0.20
25.80]
```



For loops

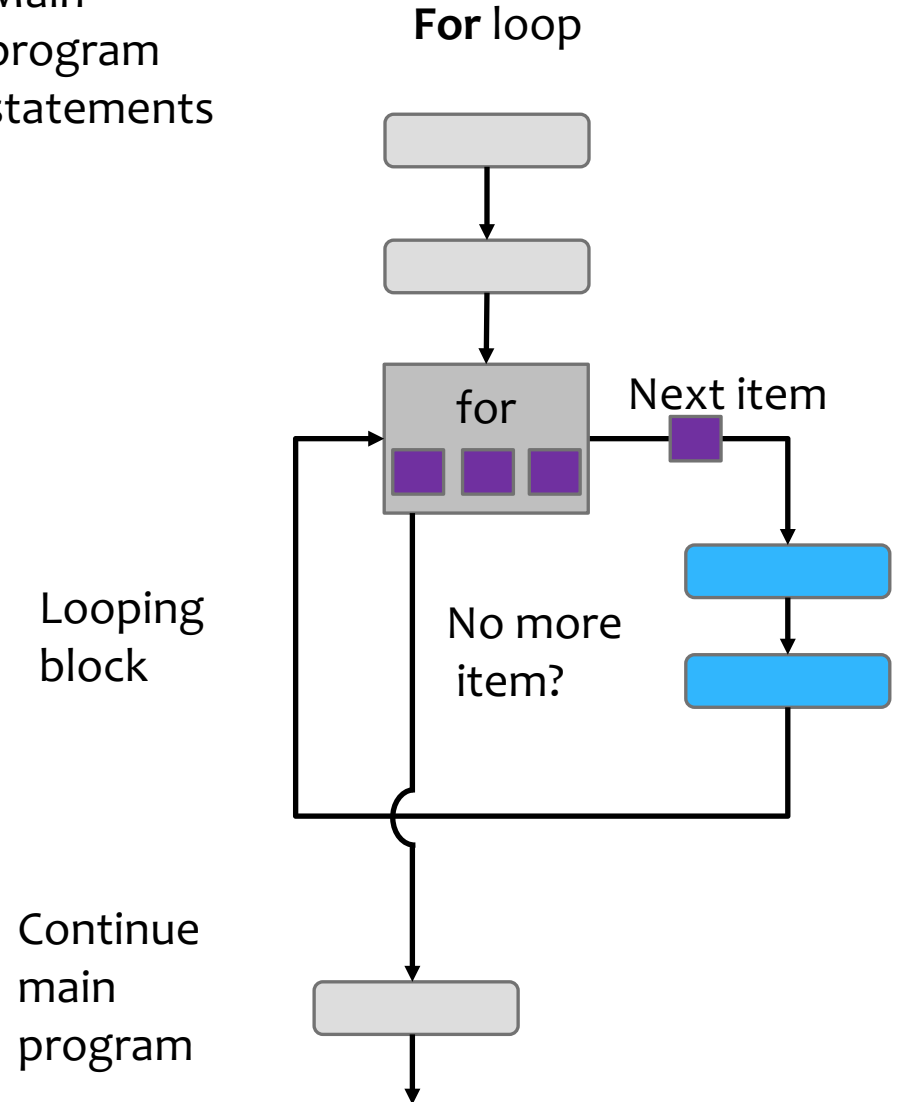
For Loops

Main program statements

- * If we want to perform the same tasks on every item in a list, string or dictionary we can use a **FOR LOOP**.

for variable in listName:

```
#any code here
```



For Loops on a List

```
[>>> myList=[4,1,3.14159]
[>>> for num in myList:
[...     print(2**num)
[...
16
2
8.824961595059897
```

(Back in the python interactive environment)

for **num** in **myList**:

 print 2^num

For Loops on a Dictionary

```
>>> myDictionary={'exp1':100,'exp2':500,'exp3':350}
>>> for key in myDictionary:
...     print('Key: %s, value: %i' % (key,myDictionary[key]))
...
Key: exp3, value: 350
Key: exp2, value: 500
Key: exp1, value: 100
```

for **key** in **myDictionary**:

 print key as a string and value as an integer

For Loops on a String

```
>>> myString='python'  
>>> myNewString=''  
>>> for letter in myString:  
...     myNewString += 2 * letter  
...  
>>> myNewString  
'ppyytthhoonn'
```

for letter in myString:

myNewString = myNewString + 2*letter

Python2 vs 3: Ranges

range(start, stop[, step])

Useful for looping over unusual ranges.

* Python2

- * **range** returns a list

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- * Use float on one of the integers for a float division

* Python3

- * **range** does not return a list

```
>>> range(10)
range(0, 10)
```

- * Can use list to see what it means

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Same syntax for the for loops

Ranges

`range(start,stop[,step])`

Useful for looping over unusual ranges.

Same syntax in Python2 and 3

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0,-10,-1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1,0,-1))
[1]
>>> myRange = range(0,10,3)
>>> for num in myRange:
...     print(num)
...
0
3
6
9
```


A Function to Find the Complement

Make a function that takes a string of nucleotides and returns a string with the reverse complement. If the string contains a character that is not a nucleotide, print a message saying so.

Pseudocode:

for nucleotide in sequence

 if nucleotide == 'A':

 prepend complementSequence with 'T'



 else if nucleotide == 'T':

Reverse Complement 2

 revComp

```
import sys
```

Import sys so we can get command line arguments

 revComp

```
import sys  
  
#prints the reverse complement of a sequence  
def reverseComp(seq):
```

Make a function that takes a sequence as an argument

A Function to Find the Complement

3

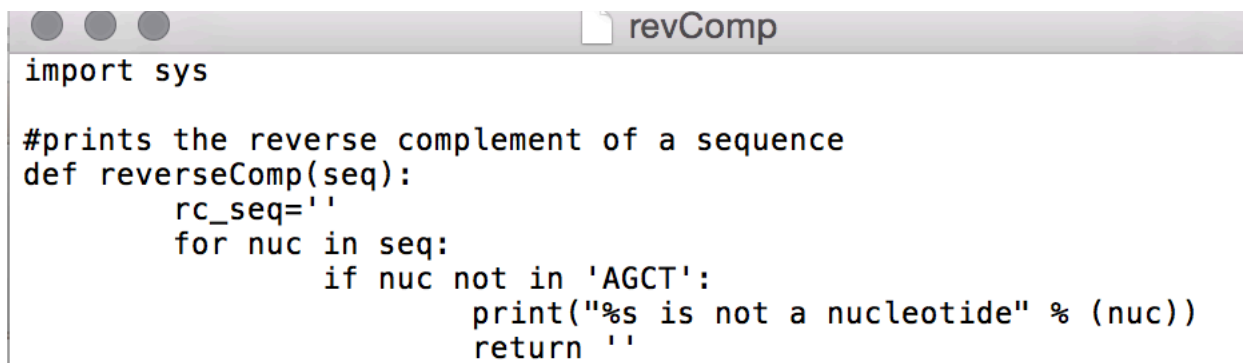


```
import sys
```

```
#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
```

Define a new empty string for the reverse complement

Use a **for loop** to do something for each nucleotide in the sequence



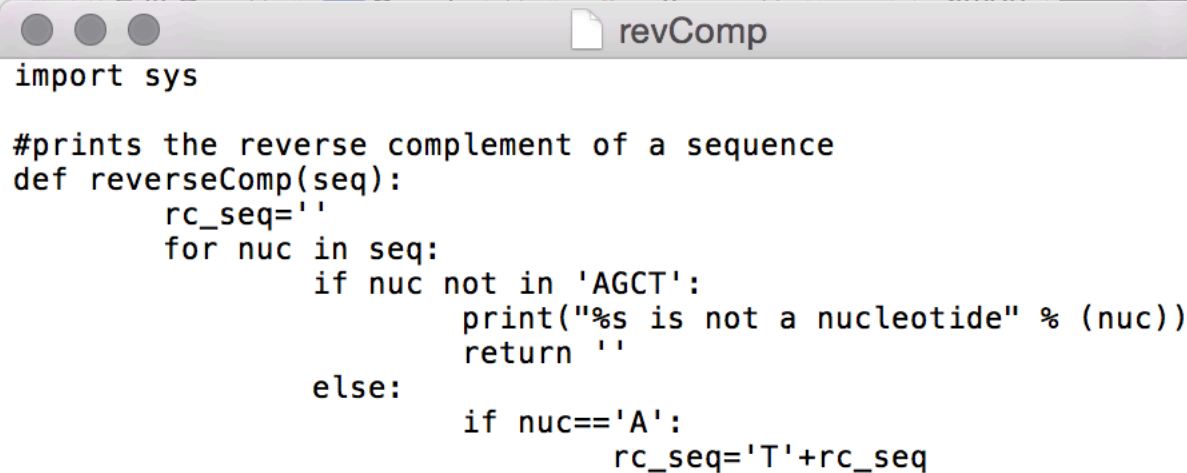
```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
```

If one of the nucleotides isn't AGCT the print a message and return nothing (quit the function without returning a new string).

A Function to Find the Complement

4



```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            if nuc=='A':
                rc_seq='T'+rc_seq
```

If the nucleotide is 'A', append T to our reverse complement string

Do the same for each nucleotide...

A Function to Find the Complement

5

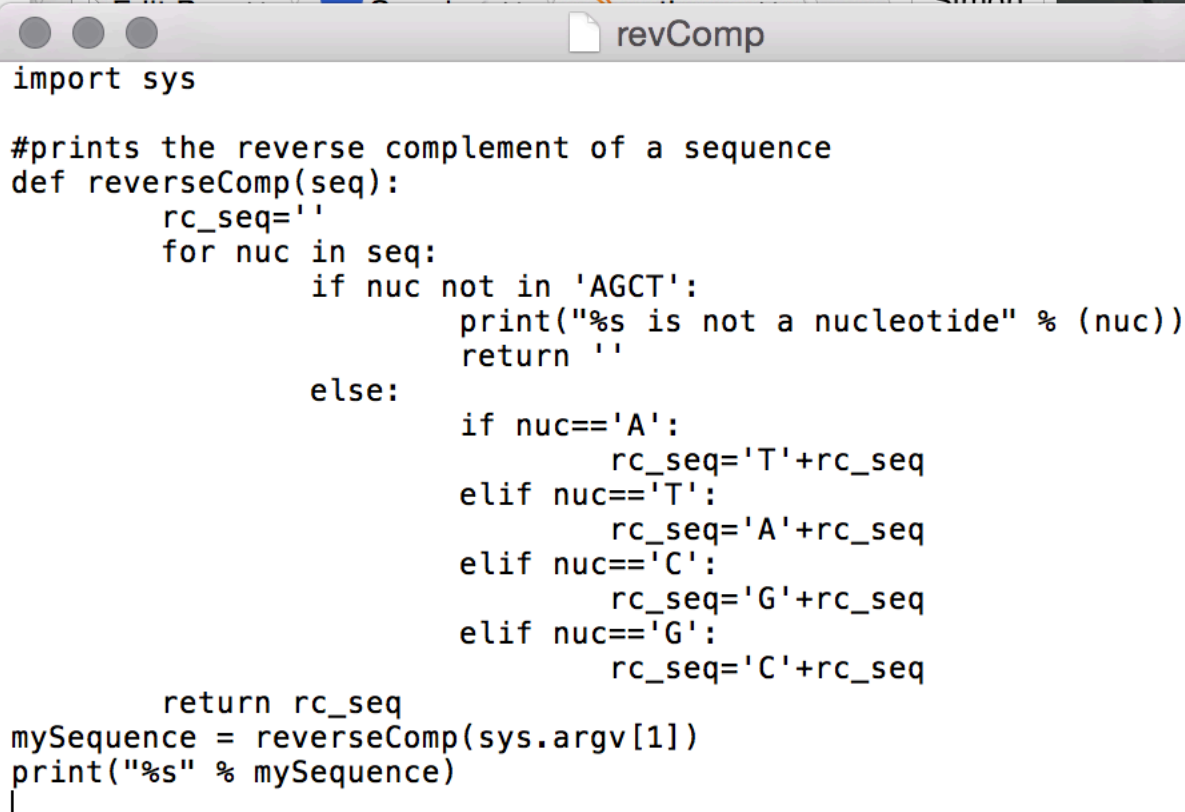
```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            if nuc=='A':
                rc_seq='T'+rc_seq
            elif nuc=='T':
                rc_seq='A'+rc_seq
            elif nuc=='C':
                rc_seq='G'+rc_seq
            elif nuc=='G':
                rc_seq='C'+rc_seq
    return rc_seq
```

The **reverseComp** function should return rc_seq string once the **for loop** has checked every nucleotide in the sequence.

A Function to Find the Complement

6



```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            if nuc=='A':
                rc_seq='T'+rc_seq
            elif nuc=='T':
                rc_seq='A'+rc_seq
            elif nuc=='C':
                rc_seq='G'+rc_seq
            elif nuc=='G':
                rc_seq='C'+rc_seq
    return rc_seq

mySequence = reverseComp(sys.argv[1])
print("%s" % mySequence)
```

Run the script and print the output.

This should be the result of passing the first command line argument to our new **reverseComp** function.

A Function to Find the Complement

7

Save it as revComp.py and let's test it!

```
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revComp.py ATTGCCTTT  
AAAGGCAAT  
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revComp.py XTTGCCTTT  
X is not a nucleotide
```

What does **return** " " used for?

```
for nuc in seq:  
    if nuc not in 'AGCT':  
        print("%s is not a nucleotide" % (nuc))
```

```
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revComp.py XTTGCCTTT  
X is not a nucleotide  
AAAGGCAA
```

```
for nuc in seq:  
    if nuc not in 'AGCT':  
        print("%s is not a nucleotide" % (nuc))  
        return ''
```

```
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revComp.py XTTGCCTTT  
X is not a nucleotide
```

A Function to Find the Complement

8

Another improvement:

```
rc_seq=  
    return rc_seq  
mySequence = reverseComp(sys.argv[1])  
print("%s" % mySequence)
```



```
rc_seq=  
    return rc_seq  
print("%s" % reverseComp(sys.argv[1]))
```

Run the function within the print statement!

A Function to Find the Complement

9

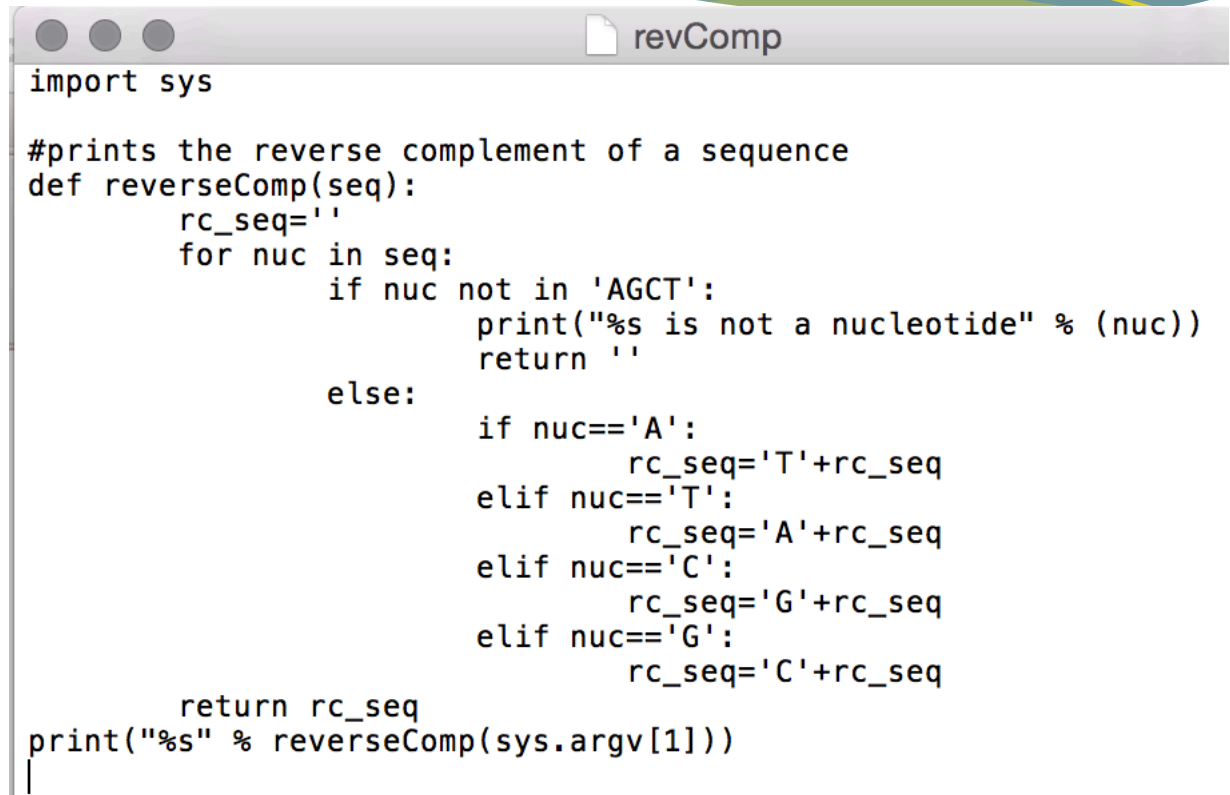
Can we make
better code than
these if
statements:

If

Elif

Elif

Elif



```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            if nuc=='A':
                rc_seq='T'+rc_seq
            elif nuc=='T':
                rc_seq='A'+rc_seq
            elif nuc=='C':
                rc_seq='G'+rc_seq
            elif nuc=='G':
                rc_seq='C'+rc_seq

    return rc_seq
print("%s" % reverseComp(sys.argv[1]))
```

Dictionaries!

```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq = ''
    compDict = {'A':'T','T':'A','C':'G','G':'C'}
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            rc_seq = compDict[nuc] + rc_seq

    return rc_seq

print("%s" % reverseComp(sys.argv[1]))
```

```
[QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revComp.py ATTGCCTTT
AAAGGCAAT
```

```
[QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revCompDictionary.py ATTGCCTTT
AAAGGCAAT
```

Works the same, much more elegant code!



More data structures

More Data Structures: Enumerate

Returns an 'enumerate' object which is the input with sequentially numbered inputs.

```
>>> seasons=['Spring','Summer','Fall','Winter']
>>> enumerate(seasons)
<enumerate object at 0x10218ec18>
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> for index,item in enumerate(seasons):
...     print(index,item)
...
0 Spring
1 Summer
2 Fall
3 Winter
```

Zip

“Zips together” two lists

```
[>>> zip([1,2,3],['a','b','c'])
<zip object at 0x10218d8c8>
[>>> list(zip([1,2,3],['a','b','c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
[>>> for x,y in zip([1,2,3],['a','b','c']):
...     print(x*y)
...
a
bb
ccc
```

Break statements

Exit the loop they are in. Notice the output isn't printed for the negative number:

```
>>> def square_root(n):  
[...     for num in n:  
[...         if num<0:  
[...             print("Can\'t take square root of negative")  
[...             break  
[...         print(num**.5)  
[...  
>>> square_root([1,4,-2,5])  
1.0  
2.0  
Can't take square root of negative
```



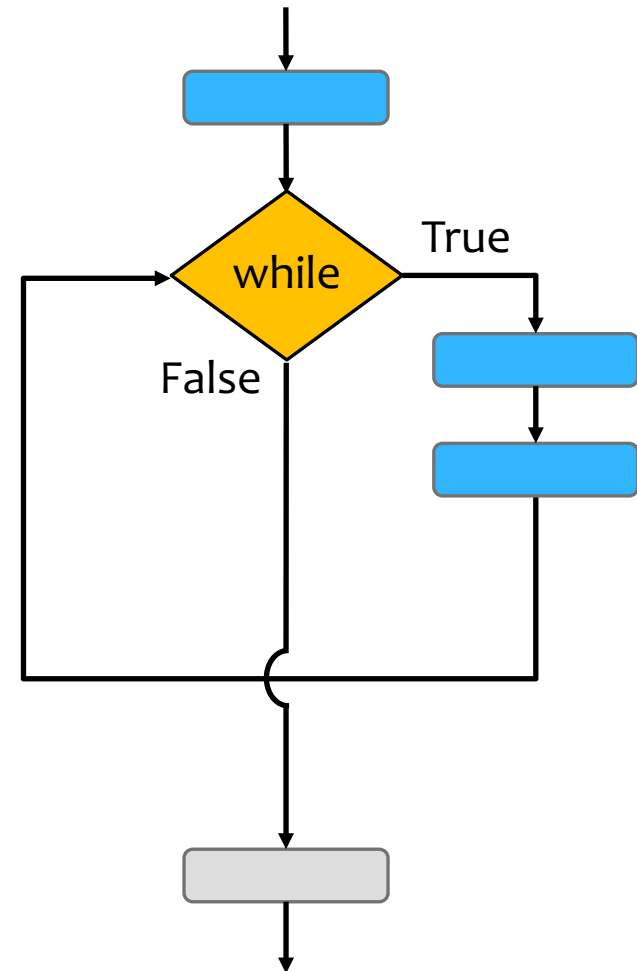
While loops

While loops

Keeps executing the code in the loop while the condition remains true.
Rechecks the condition after each iteration.

while condition:

#code to execute



While loops




```
>>> loopCondition=True
>>> while loopCondition:
...     loopCondition=False
...     print("This will print once")
...
This will print once
```

Set loopCondition to True.

While loop checks if loopCondition is true.

It is, so the code inside the loop will be executed next.

While loops




```
>>> loopCondition=True
>>> while loopCondition:
...     loopCondition=False
...     print("This will print once")
...
This will print once
```

Set loopCondition to False.

The while loop doesn't recheck the loopCondition until it reaches the end so the code will continue executing.

While loops



```
>>> loopCondition=True
>>> while loopCondition:
...     loopCondition=False
...     print("This will print once")
...
This will print once
```

Print “this will print once”.

We are at the end of the loop now so the loopCondition will be checked next.

While loops



```
>>> loopCondition=True
>>> while loopCondition:
...     loopCondition=False
...     print("This will print once")
...
This will print once
```

loopCondition is False now so the code inside the loop will not be executed.

While loops

```
>>> loopCondition=True
>>> while loopCondition:
...     loopCondition=False
...     print("This will print once")
...
This will print once
```

Indeed the text is printed just once!

While loops

Don't forget to include the `count+=1` else you create an infinite loop!

Why does it print 9 last yet
`count = 10` after the code is finished?

How do we get it to print all the way
to 10?

```
[>>> count = 0
[>>> while count < 10:
[...     print(count)
[...     count += 1
[...
0
1
2
3
4
5
6
7
8
9
[>>> count
10
```

While loops

Switching order of count and print statements is one way!

Could also have made condition:
While count <=10

```
[>>> count = 0
[>>> while count <= 10:
[...     print(count)
[...     count += 1
[...
0
1
2
3
4
5
6
7
8
9
10
[>>> count
11
```

While loops

Keep doing a loop until the correct input is received:

```
>>> choice = ''
>>> while choice != 'python':
...     choice = input('What are we learning? ')
...
What are we learning? Java
What are we learning? HTML
What are we learning? pthon
What are we learning? python
>>> |
```

Reminder: in Python2, it should be **raw_input**

Break statements can exit while loops

The while loop condition is never met but the code reaches a break before count reaches 100.

```
>>> count = 0
>>> while count < 100:
...     print(count)
...     count += 1
...     if 100 / count < 10:
...         break
...
0
1
2
3
4
5
6
7
8
9
10
>>> |
```

While / Else

Else: only executed if while loop finishes without reaching a break.

```
[>>> from random import randrange
>>> def randomNumberGame():
...     count = 0
...     random_number=randrange(1,10)
...     while count<3:
...         guess=int(input("Guess a number"))
...         if guess==random_number:
...             print("YOU WIN!")
...             break
...         count += 1
...     else:
...         print("You lost.")
[...
[...
```

Play the random number game!

```
>>> from random import randrange
>>> def randomNumberGame():
...     count = 0
...     random_number=randrange(1,10)
...     while count<3:
...         guess=int(input("Guess a number"))
...         if guess==random_number:
...             print("YOU WIN!")
...             break
...         count += 1
...     else:
...         print("You lost.")
[...
[...
```

```
[Guess a number3
```

```
[Guess a number4
```

```
YOU WIN!
```

Reverse Complement Returns

```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    compDict = {'A':'T','T':'A','C':'G','G':'C'}
    for nuc in seq:
        if nuc not in 'AGCT':
            print("%s is not a nucleotide" %
(nuc))
            return ''
        else:
            rc_seq = compDict[nuc] + rc_seq
    return rc_seq
print("%s" % reverseComp(sys.argv[1]))
```

```
import sys

#prints the reverse complement of a sequence
def reverseComp(seq):
    rc_seq=''
    compDict = {'A':'T','T':'A','C':'G','G':'C'}
    count=0
    while count < len(seq):
        if seq[count] not in 'AGCT':
            print("%s is not a nucleotide" % (nuc))
            return ''
        else:
            rc_seq = compDict[seq[count]] + rc_seq
            count+=1
    return rc_seq
print("%s" % reverseComp(sys.argv[1]))
```



Errors and exceptions

Errors

Everyone gets errors in their code. You may already have had some!

Knowing what the errors mean help you fix them.

Errors messages are quite informative even if they seem difficult to understand

```
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revCompDictionary.py ATTGCCTTT  
AAAGGCAAT  
QCBs-MacBook-Pro:~ qcbcollaboratory$ python3 revCompWhile.py ATTGCCTTT  
AAAGGCAAT
```

Syntax Error

```
>>> while True
      File "<stdin>", line 1
        while True
                ^
SyntaxError: invalid syntax
```

Notice the error highlighting which part of the code is incorrect. Syntax errors are the most generic and common.

To fix, check the line in the error message, specifically check around the arrow.

What is wrong with the first line above?

Indentation Error

```
[>>> while True:
[... print('test')
      File "<stdin>", line 2
        print('test')
          ^
IndentationError: expected an indented block
```

We've fixed the while True: line.

Indentation error is a specific type of syntax error which tells you your code was not correctly indented.

How do we correct this code?

Exceptions

Sometimes code will be valid and won't cause an error while you input it but can error when it is executed.

Errors that occur at the time code runs are called **exceptions**.

Not all exceptions are fatal, you can include code to handle exceptions.

Exceptions

Name error

```
>>> someText
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'someText' is not defined
```

Value error

```
[>>> int('aaa')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'aaa'
```

Divide by
zero error

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Keyboard
interrupt
(ctrl+c)

```
>>>
KeyboardInterrupt
```

Exceptions

Type error

```
>>> len(123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Input object
error

```
>>> open('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: ''
```

Let's figure out how to handle these exceptions...

Handling Exceptions

```
>>> while True:
...     x=int(input("Please enter a number: "))
...     break
...
Please enter a number: a
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

We can see that this code throws a **ValueError**.

If we don't want this to stop the program, or we want to show a more helpful error message then we need to add some code:

Handling Exceptions

```
>>> while True:
...     try:
...         x=int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Not a valid number. Try again.")
...
Please enter a number: a
Not a valid number. Try again.
Please enter a number: b
Not a valid number. Try again.
Please enter a number: 1
```

The **try** section is executed first.

If a number is received, then no exception will be thrown so the **break** command will be reached

Handling Exceptions

```
>>> while True:
...     try:
...         x=int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Not a valid number. Try again.")
...
[Please enter a number: a
Not a valid number. Try again.
Please enter a number: b
Not a valid number. Try again.
Please enter a number: 1
```

If no error types are matched the code will throw an unformatted exception as if the **try** and **except** commands were not there.

Handling Exceptions

```
>>> while True:
...     try:
...         x=int(input("Please enter a number: "))
...     except (TypeError, ValueError):
...         print("Not a valid number. Try again.")
...     except (KeyboardInterrupt):
...         print("Attempted to end input")
...     else:
...         print("Goods job! You picked a number.")
...         break
...
Please enter a number: Attempted to end input
Please enter a number: k
Not a valid number. Try again.
Please enter a number: 3
Goods job! You picked a number.
```

Can have multiple exceptions handled by the same section.
Have an **else** clause that runs if the try ends without a **break** command.

Exception Hierarchy

If you handle a class it will handle all subclasses, so consider that if you catch **StandardError** it will be difficult to write code to handle all possible exceptions. Try and handle as low level exception as possible and avoid:
except Exception

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- OSError
        |   +-- BlockingIOError
```