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

```python
>>> myList = []
>>> myEmptyList=[]
>>> myShoppingList=["Apples", "Beer", "Chocolate"]
>>> myShoppingList
['Apples', 'Beer', 'Chocolate']
>>> myMixedList = [123, "a string", 2.75]
>>> myMixedList
[123, 'a string', 2.75]
```
* Can access and change elements of a list by index.
* Starting at 0
* myList[0]
* Just like strings.

```python
>>> myNucList=['A','G','C','T']
>>> myNucList
['A', 'G', 'C', 'T']
>>> myNucList[2]
'C'
>>> myNucList[2]='G'
>>> myNucList[2]
'G'
>>> myNucList
['A', 'G', 'G', 'T']
```
* 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', 'A', 'C', 'T', 'C']
>>> myNucList.sort()
>>> myNucList
['A', 'A', 'C', 'C', 'G', 'T']
>>> myNucList.remove('G')
>>> myNucList
['A', 'A', 'C', 'C', 'T']
>>> del myNucList[1]
>>> myNucList
['A', 'C', 'C', 'T']
>>> 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]
[]
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!

```python
def experiments:
    'control': 1000,
    'exp1': 500,
    'exp2': 600,

>>> experiments['control']
1000
>>> experiments['exp2']
600
>>> experiments['exp1']
500
>>> experiments['exp3'] = 300
```

```python
>>> del experiments['exp2']
```
* 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

```python
# 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

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

```
SiMac:~ simon$ python 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

```
import sys
for arg in sys.argv:
    print(arg)

# 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)
```

```
SiMac:~ simon$ python tipCalculator.py arg1 arg2 arg3
```

Sys.argv[0] is tipCalculator.py
Sys.argv[1] arg1...
```python
import sys
for arg in sys.argv:
    print(arg)

# 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)
```

```python
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

- 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.

```python
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]))
```
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]))

SiMac:~ simon$ python tipCalculator.py 20.00 0.15
24.73
SiMac:~ simon$ python tipCalculator.py 20.00 0.20
25.80
* If we want to perform the same tasks on every item in a list, string or dictionary we can use a **FOR LOOP**.

```python
for variable in listName:
    #any code here
```
For Loops on a List

(Back in the python interactive environment)

For **num** in **myList**:  
    print \(2^{\text{num}}\)
for key in myDictionary:
    print key as a string and value as an integer
For Loops on a String

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

>>> myNewString
'ppyytththhoonn'
```

for letter in myString:
    myNewString = myNewString + 2*letter
**Ranges**

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

Useful for looping over unusual ranges.
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’:
        ....
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

Define a new empty string for the reverse complement

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

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

```
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))
        else:
            if nuc == 'A':
                rc_seq = 'T' + rc_seq
    return rc_seq

# Example usage
seq = 'AGCT

print(reverseComp(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

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

```python
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
```
A Function to Find the Complement

```python
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

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

```
SiMac:~ simon$ python revComp.py ATTCGCTTT AAAGGCAAAT
SiMac:~ simon$ python revComp.py XTTGCCTTT
X is not a nucleotide
AAAGGCAA
```

What if we want to stop if an incorrect character is found?

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

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

```
SiMac:~ simon$ python revComp.py XTTGCCTTT
X is not a nucleotide
```
A Function to Find the Complement

Another improvement:

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

Run the function within the print statement!
A Function to Find the Complement

Can we make better code than these if statements:

If
Elif
Elif
Elif

```python
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!

```python
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))
        else:
            rc_seq = compDict[nuc] + rc_seq
    return rc_seq

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

Works the same, much more elegant code!
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 0x10395ea50>
>>> 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')
```
“Zips together” two lists

```python
>>> zippedList = zip([1,2,3],['a','b','c'])
>>> zippedList
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> for x,y in zippedList:
...     print(x*y)
...     a
...     bb
...     ccc
```
Break statements

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

```python
>>> def square_root(n):
...     for num in n:
...         if num<0:
...             print("Can\'t take squareroot of a negative")
...             break
...         print num**.5
...     print square_root([1,4,5,9])
1.0
2.0
2.2360679775
3.0
```
While loops

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

**while** condition:

```python
# code to execute
```
While loops

Set `loopCondition` to True.
While loop checks if `loopCondition` is true.
It is, so the code inside the loop will be executed next.
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

Print “this will print once”.
We are at the end of the loop now so the loopCondition will be checked next.
While loops

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

Indeed the text is printed just once!

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

Switching order of count and print statements is one way!

Could also have made condition:
While count <=10

```python
>>> count = 0
>>> while count < 10:
...    count += 1
...    print count
...
1
2
3
4
5
6
7
8
9
10
```
While loops

Keep doing a loop until the correct input is received:

```python
>>> choice = ''
>>> while choice != 'python':
...    choice = raw_input('What language are we learning?')
...
What language are we learning? Java
What language are we learning? HTML
What language are we learning? Python
What language are we learning? Python
```
Break statements can exit while loops

The while loop condition is never met but the code reaches a break before `count` reaches 100.
Else: only executed if while loop finishes without reaching a break.

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

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

>>> randomNumberGame()
Guess a number: 6
Guess a number: 5
Guess a number: 4
YOU WIN!
```
Reverse Complement Returns

import sys

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

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]))
```python
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]))
```

```
SiMac:~ simon$ python revCompDictionary.py ATTGCCTTTAAAGGCCAAT
SiMac:~ simon$ python revCompWhile.py ATTGCCTTTAAAGGCCAAT
```
Everyone gets errors in their code. You may already have had some!

Knowing what the errors mean helps you fix them.

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

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?

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

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

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

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

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

We can see that this code throws a `ValueError`.
If we don’t want this to stop the program, or we want show a more helpful error message then we need to add some code:
Handling Exceptions

The `try` section is executed first.

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

```python
>>> while True:
...     try:
...         x=int(raw_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
```
Handling Exceptions

If an exception occurs anywhere the code immediately stop running the try statement and tries to match the exception thrown with an `except` and then runs the code inside the matching except clause.
Handling Exceptions

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

```python
>>> while True:
...     try:
...         x=int(raw_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
```
Handling Exceptions

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

```python
except Exception
```