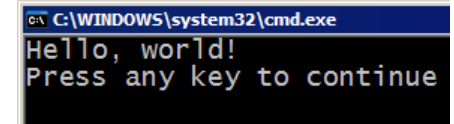
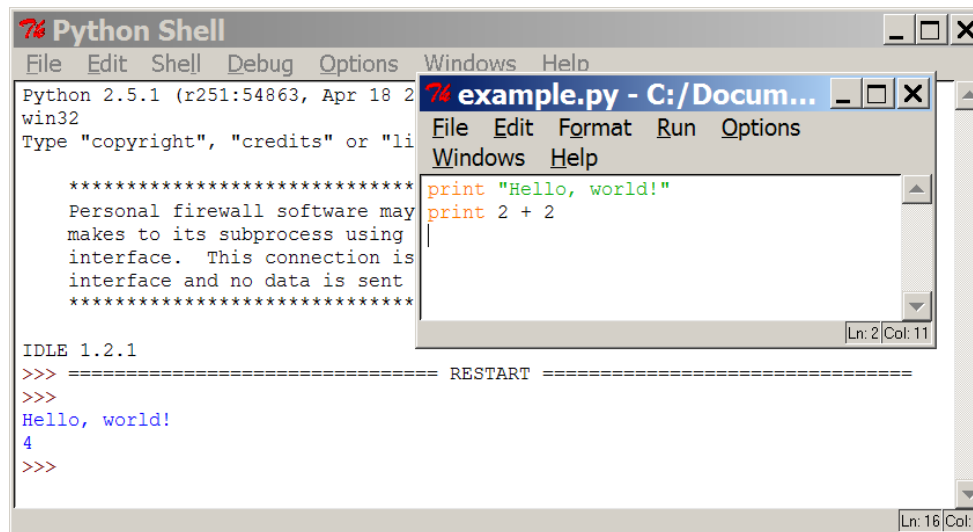


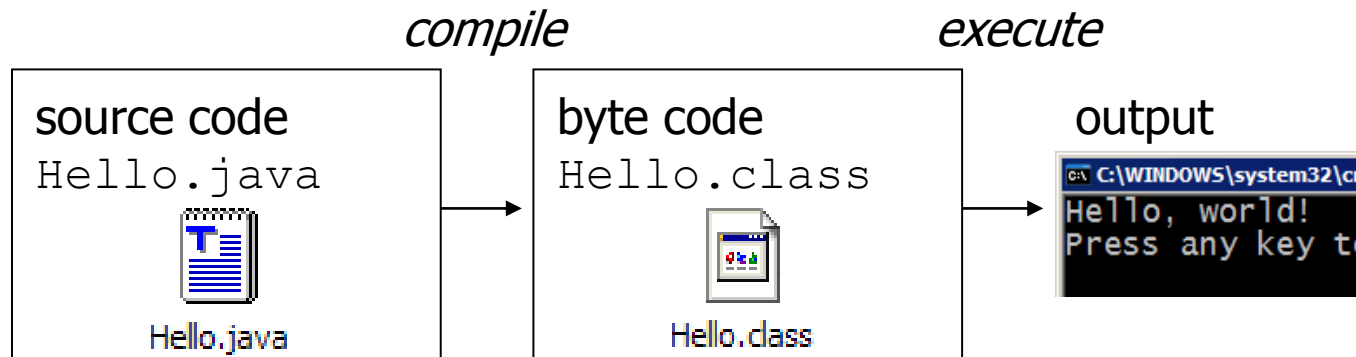
Programming basics

- **code** or **source code**: The sequence of instructions in a program.
- **syntax**: The set of legal structures and commands that can be used in a particular programming language.
- **output**: The messages printed to the user by a program.
- **console**: The text box onto which output is printed.
 - Some source code editors pop up the console as an external window, and others contain their own console window.

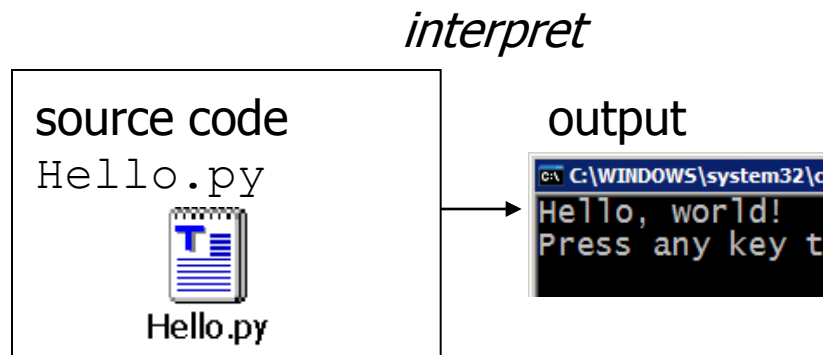


Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



- Python is instead directly *interpreted* into machine instructions.



The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

Expressions

- **expression:** A data value or set of operations to compute a value.

Examples: $1 + 4 * 3$
 42

- Arithmetic operators we will use:

$+$	$-$	$*$	$/$	addition, subtraction/negation, multiplication, division
$\%$				modulus, a.k.a. remainder
$**$				exponentiation

- **precedence:** Order in which operations are computed.

- $*$ $/$ $\%$ $**$ have a higher precedence than $+$ $-$

$1 + 3 * 4$ is 13

- Parentheses can be used to force a certain order of evaluation.

$(1 + 3) * 4$ is 16

Integer division

- When we divide integers with $/$, the quotient is also an integer.

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 52 \\ 27 \overline{) 1425} \\ \underline{135} \\ 75 \\ \underline{54} \\ 21 \end{array}$$

- More examples:

- $35 / 5$ is 7
- $84 / 10$ is 8
- $156 / 100$ is 1

- The $\%$ operator computes the remainder from a division of integers.

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 43 \\ 5 \overline{) 218} \\ \underline{20} \\ 18 \\ \underline{15} \\ 3 \end{array}$$

Real numbers

- Python can also manipulate real numbers.
 - Examples: 6.022 -15.9997 42.0 2.143e17
- The operators + - * / % ** () all work for real numbers.
 - The / produces an exact answer: 15.0 / 2.0 is 7.5
 - The same rules of precedence also apply to real numbers:
Evaluate () before * / % before + -
- When integers and reals are mixed, the result is a real number.
 - Example: 1 / 2.0 is 0.5
 - The conversion occurs on a per-operator basis.

$$\begin{array}{rcl} 7 / 3 * 1.2 + 3 / 2 & & \\ \underline{2} * 1.2 + 3 / 2 & & \\ 2.4 + 3 / 2 & & \\ 2.4 + \underline{1} & & \\ 3.4 & & \end{array}$$

Math commands

- Python has useful commands (or called functions) for performing calculations.

Command name	Description
<code>abs(value)</code>	absolute value
<code>ceil(value)</code>	rounds up
<code>cos(value)</code>	cosine, in radians
<code>floor(value)</code>	rounds down
<code>log(value)</code>	logarithm, base e
<code>log10(value)</code>	logarithm, base 10
<code>max(value1, value2)</code>	larger of two values
<code>min(value1, value2)</code>	smaller of two values
<code>round(value)</code>	nearest whole number
<code>sin(value)</code>	sine, in radians
<code>sqrt(value)</code>	square root

Constant	Description
e	2.7182818...
pi	3.1415926...

- To use many of these commands, you must write the following at the top of your Python program:

```
from math import *
```

Numbers: Floating Point

- `int(x)` converts `x` to an integer
- `float(x)` converts `x` to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.23232000000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```


Variables

- **variable:** A named piece of memory that can store a value.

- Usage:

- Compute an expression's result,
- store that result into a variable,
- and use that variable later in the program.



- **assignment statement:** Stores a value into a variable.

- Syntax:

name = value

- Examples:

$x = 5$

$\text{gpa} = 3.14$

x 5

gpa 3.14

- A variable that has been given a value can be used in expressions.

$x + 4$ is 9

- **Exercise:** Evaluate the quadratic equation for a given a , b , and c .

Example

```
>>> x = 7
>>> x
7
>>> x+7
14
>>> x = 'hello'
>>> x
'hello'
>>>
```

print

- `print` : Produces text output on the console.

- Syntax:

```
print "Message"
```

```
print Expression
```

- Prints the given text message or expression value on the console, and moves the cursor down to the next line.

```
print Item1, Item2, ..., ItemN
```

- Prints several messages and/or expressions on the same line.

- Examples:

```
print "Hello, world!"
```

```
age = 45
```

```
print "You have", 65 - age, "years until retirement"
```

Output:

```
Hello, world!
```

```
You have 20 years until retirement
```

Example: print Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (`print 'hello',`) will not print a newline character

```
>>> print 'hello'
```

```
hello
```

```
>>> print 'hello', 'there'
```

```
hello there
```

input

- `input` : Reads a number from user input.
 - You can assign (store) the result of `input` into a variable.
 - Example:

```
age = input("How old are you? ")
print "Your age is", age
print "You have", 65 - age, "years until retirement"
```

Output:

```
How old are you? 53
Your age is 53
You have 12 years until retirement
```

- **Exercise:** Write a Python program that prompts the user for his/her amount of money, then reports how many Nintendo Wiis the person can afford, and how much more money he/she will need to afford an additional Wii.

Input: Example

```
print "What's your name?"
```

```
name = raw_input("> ")
```

```
print "What year were you born?"
```

```
birthyear = int(raw_input("> "))
```

```
print "Hi ", name, "!", "You are ", 2016 - birthyear
```

```
% python input.py
```

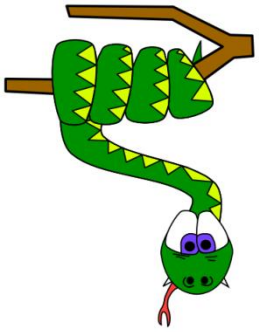
```
What's your name?
```

```
> Michael
```

```
What year were you born?
```

```
> 1980
```

```
Hi Michael! You are 31
```



Repetition (loops) and Selection (if/else)

The for loop

- **for loop**: Repeats a set of statements over a group of values.

- Syntax:

```
for variableName in groupOfValues:  
    statements
```

- We indent the statements to be repeated with tabs or spaces.
- **variableName** gives a name to each value, so you can refer to it in the **statements**.
- **groupOfValues** can be a range of integers, specified with the `range` function.

- Example:

```
for x in range(1, 6):  
    print x, "squared is", x * x
```

Output:

```
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25
```


range

- The `range` function specifies a range of integers:
 - `range(start, stop)` - the integers between **start** (inclusive) and **stop** (exclusive)
 - It can also accept a third value specifying the change between values.
 - `range(start, stop, step)` - the integers between **start** (inclusive) and **stop** (exclusive) by **step**

- Example:

```
for x in range(5, 0, -1):  
    print x  
print "Blastoff!"
```

Output:

```
5  
4  
3  
2  
1  
Blastoff!
```

- **Exercise:** How would we print the "99 Bottles of Beer" song?

Cumulative loops

- Some loops incrementally compute a value that is initialized outside the loop. This is sometimes called a *cumulative sum*.

```
sum = 0
for i in range(1, 11):
    sum = sum + (i * i)
print "sum of first 10 squares is", sum
```

Output:

```
sum of first 10 squares is 385
```

- **Exercise:** Write a Python program that computes the factorial of an integer.

if

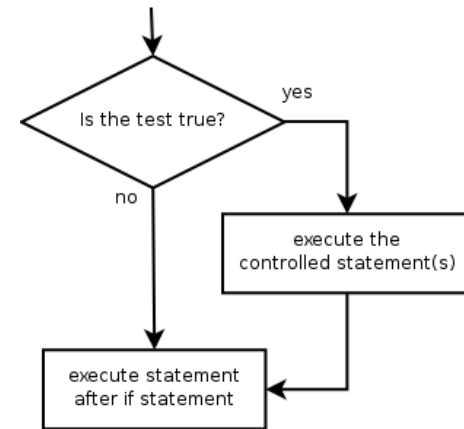
- **if statement:** Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.

- Syntax:

```
if condition:  
    statements
```

- Example:

```
gpa = 3.4  
if gpa > 2.0:  
    print "Your application is accepted."
```



if/else

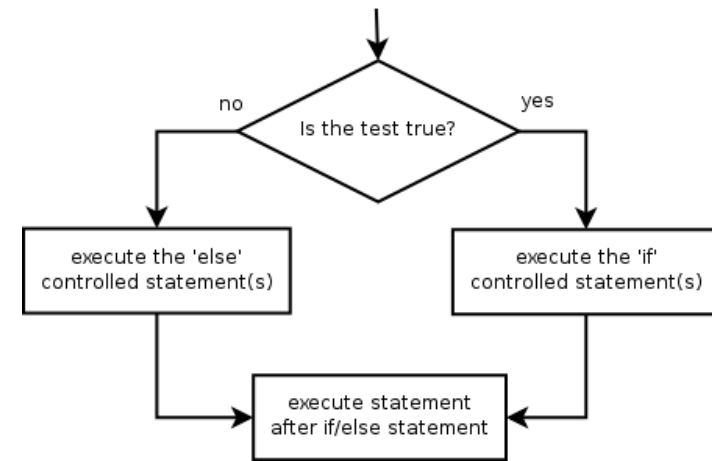
- **if/else statement:** Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

- Syntax:

```
if condition:  
    statements  
else:  
    statements
```

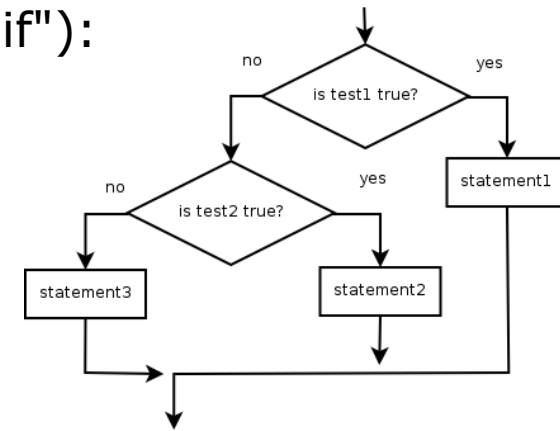
- Example:

```
gpa = 1.4  
if gpa > 2.0:  
    print "Welcome to Mars University!"  
else:  
    print "Your application is denied."
```



- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:  
    statements  
elif condition:  
    statements  
else:  
    statements
```



Example of If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30 :
    y = x + 30
else :
    y = x
print `y = `,
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y = 0.999911860107
>>>
```

In interpreter

while

- **while loop:** Executes a group of statements as long as a condition is True.
 - good for *indefinite loops* (repeat an unknown number of times)

- **Syntax:**

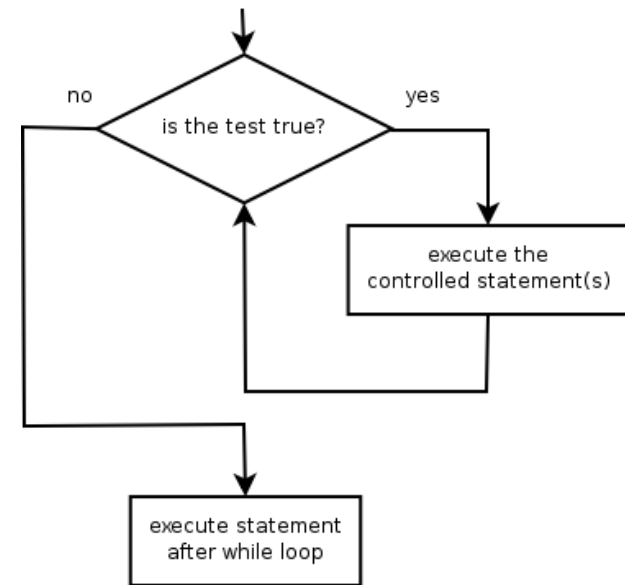
```
while condition:  
    statements
```

- **Example:**

```
number = 1  
while number < 200:  
    print number,  
    number = number * 2
```

- **Output:**

1 2 4 8 16 32 64 128



While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

■ In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

■ In interpreter

Logic

- Many logical expressions use *relational operators*:

Operator	Meaning	Example	Result
==	equals	<code>1 + 1 == 2</code>	True
!=	does not equal	<code>3.2 != 2.5</code>	True
<	less than	<code>10 < 5</code>	False
>	greater than	<code>10 > 5</code>	True
<=	less than or equal to	<code>126 <= 100</code>	False
>=	greater than or equal to	<code>5.0 >= 5.0</code>	True

- Logical expressions can be combined with *logical operators*:

Operator	Example	Result
and	<code>9 != 6 and 2 < 3</code>	True
or	<code>2 == 3 or -1 < 5</code>	True
not	<code>not 7 > 0</code>	False

- Exercise:** Write code to display and count the factors of a number.

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

More Examples For Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in [1,7,13,2]:  
    print x
```

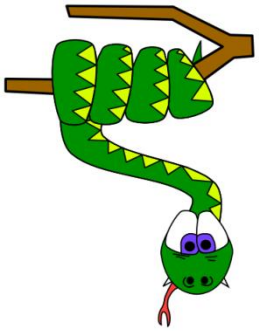
```
%python forloop1.py  
1  
7  
13  
2
```

forloop2.py

```
for x in range(5) :  
    print x
```

```
% python forloop2.py  
0  
1  
2  
3  
4
```

range(N) generates a list of numbers [0,1, ..., n-1]



More Data Types

Everything is an object

- Everything means everything, including functions and classes (more on this later!)
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 **
2
17509376329L
>>>
```

Numbers: Floating Point

- `int(x)` converts `x` to an integer
- `float(x)` converts `x` to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.23232000000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

String Literals

- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```


String Literals

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
```

```
'I am a string'
```

```
>>> "So am I!"
```

```
'So am I!'
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String
- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Types for Data Collection

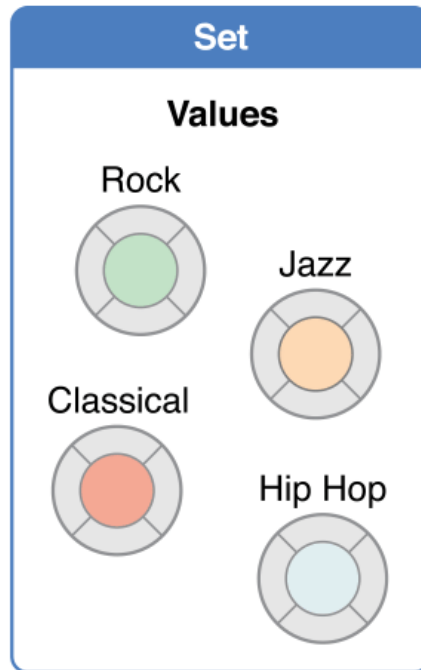
List, Set, and Dictionary

List

Indexes	Values
0	Six Eggs
1	Milk
2	Flour
3	Baking Powder
4	Bananas

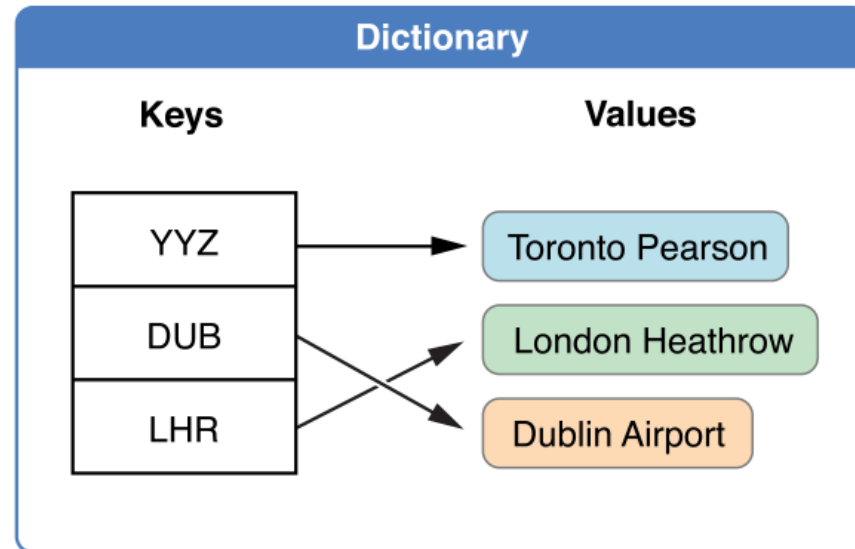
■ Ordered

Set



■ Unordered list

Dictionary



Pairs of values

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

List Functions

- `list.append(x)`
 - Add item at the end of the list.
- `list.insert(i,x)`
 - Insert item at a given position.
 - Similar to `a[i:i]=[x]`
- `list.remove(x)`
 - Removes first item from the list with value x
- `list.pop(i)`
 - Remove item at position I and return it. If no index I is given then remove the first item in the list.
- `list.index(x)`
 - Return the index in the list of the first item with value x.
- `list.count(x)`
 - Return the number of time x appears in the list
- `list.sort()`
 - Sorts items in the list in ascending order
- `list.reverse()`
 - Reverses items in the list

Lists: Modifying Content

- **x[i] = a** reassigns the *i*th element to the value *a*
- Since *x* and *y* point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```


Using Lists as Stacks

- You can use a list as a stack

```
>>> a = ["a", "b", "c", "d"]
```

```
>>> a
```

```
['a', 'b', 'c', 'd']
```

```
>>> a.append("e")
```

```
>>> a
```

```
['a', 'b', 'c', 'd', 'e']
```

```
>>> a.pop()
```

```
'e'
```

```
>>> a.pop()
```

```
'd'
```


```
>>> a = ["a", "b", "c"]
```

```
>>>
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:
' ,' is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```



Sets

- A set is another python data structure that is an unordered collection with no duplicates.

```
>>> setA=set(["a","b","c","d"])
```

```
>>> setB=set(["c","d","e","f"])
```

```
>>> "a" in setA
```

```
True
```

```
>>> "a" in setB
```

```
False
```

Sets

```
>>> setA - setB
```

```
{'a', 'b'}
```

```
>>> setA | setB
```

```
{'a', 'c', 'b', 'e', 'd', 'f'}
```

```
>>> setA & setB
```

```
{'c', 'd'}
```

```
>>> setA ^ setB
```

```
{'a', 'b', 'e', 'f'}
```

```
>>>
```



Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d= {'one' : 1, 'two' : 2, 'three' : 3}  
>>> d['three']  
3
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```


Iterating over a dictionary

```
>>> address={'Wayne': 'Young 678', 'John': 'Oakwood 345',  
            'Mary': 'Kingston 564'}  
>>> for k in address.keys():  
        print(k,":", address[k])
```

```
Wayne : Young 678  
John : Oakwood 345  
Mary : Kingston 564  
>>>
```

```
>>> for k in sorted(address.keys()):  
    print(k,":", address[k])
```

```
John : Oakwood 345  
Mary : Kingston 564  
Wayne : Young 678  
>>>
```



Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

Integers: 2323, 3234L

Floating Point: 32.3, 3.1E2

Complex: 3 + 2j, 1j

Lists: l = [1,2,3]

Tuples: t = (1,2,3)

Dictionaries: d = {'hello' : 'there', 2 : 15}

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references