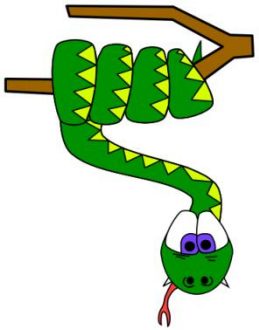
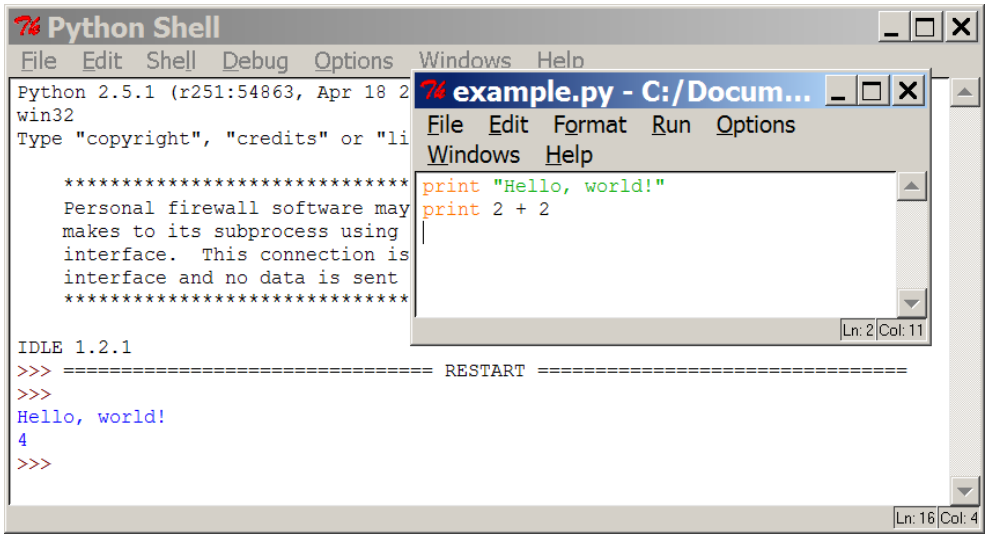


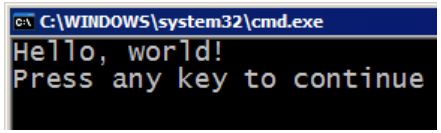

# Introduction to Programming with Python

Python Review. Modified slides from Marty Stepp and Moshe Goldstein

# Programming basics

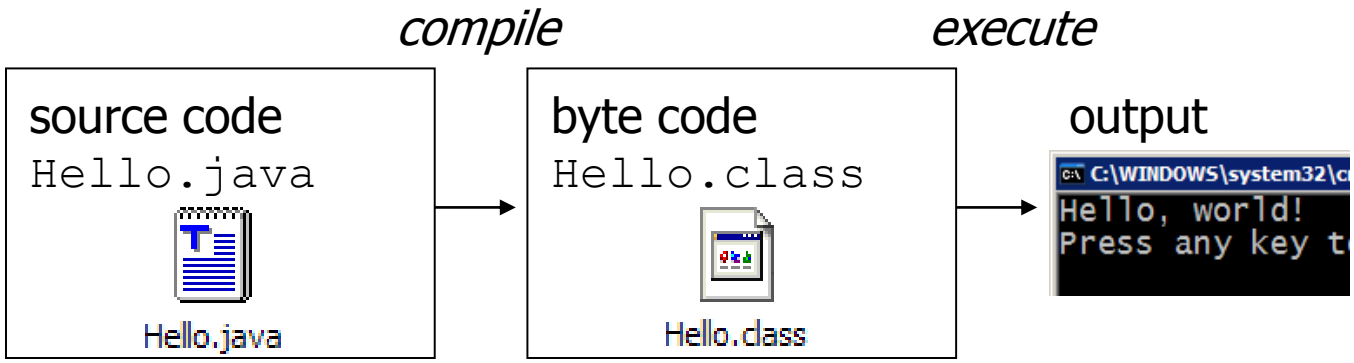
- **code** or **source code**: The sequence of instructions in a program.
- **syntax**: The set of legal structures and commands that can be used in a particular programming language.
- **output**: The messages printed to the user by a program.
- **console**: The text box onto which output is printed.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



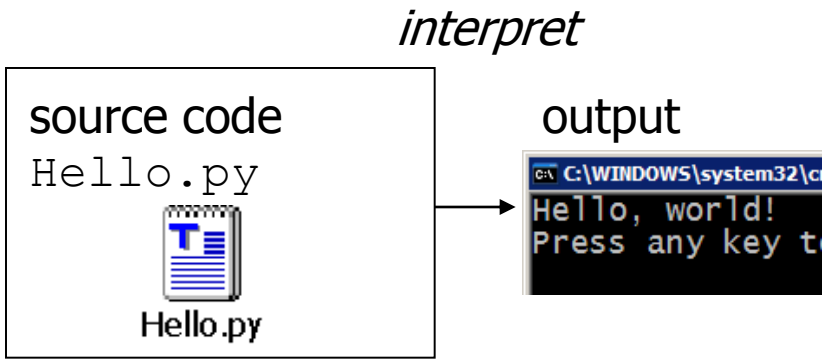



# Compiling and interpreting

- 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.

```
    3
4 ) 14
    12
    --
     2
```

```
   52
27 ) 1425
   135
   ---
    75
    54
    --
    21
```

  - More examples:
    - `35 / 5` is `7`
    - `84 / 10` is `8`
    - `156 / 100` is `1`

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

```
    3
4 ) 14
    12
    --
     2
```

```
   43
5 ) 218
   20
   --
    18
    15
    --
     3
```

# 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.

```
7 / 3 * 1.2 + 3 / 2
  2   * 1.2 + 3 / 2
    2.4     + 3 / 2
    2.4     +   1
        3.4
```

# Math commands

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

| Command name | Description |
|---|---|
| `abs(`***value***`)` | absolute value |
| `ceil(`***value***`)` | rounds up |
| `cos(`***value***`)` | cosine, in radians |
| `floor(`***value***`)` | rounds down |
| `log(`***value***`)` | logarithm, base *e* |
| `log10(`***value***`)` | logarithm, base 10 |
| `max(`***value1***`, `***value2***`)` | larger of two values |
| `min(`***value1***`, `***value2***`)` | smaller of two values |
| `round(`***value***`)` | nearest whole number |
| `sin(`***value***`)` | sine, in radians |
| `sqrt(`***value***`)` | 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
1.2323200000000001
>>> 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
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.

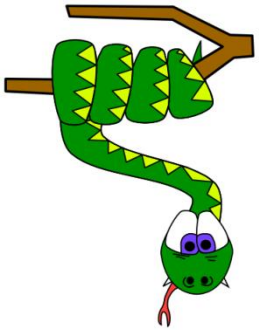
# Input: Example

```
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.

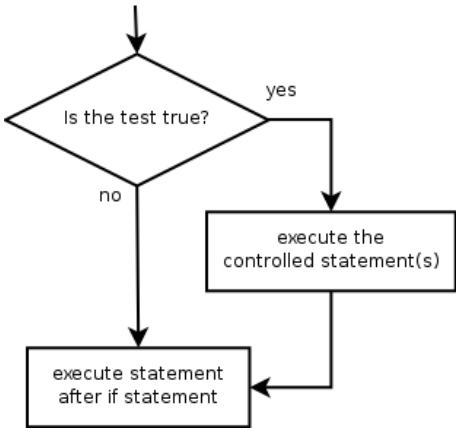
# if

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

```
if condition:
    statements
```

- Example:

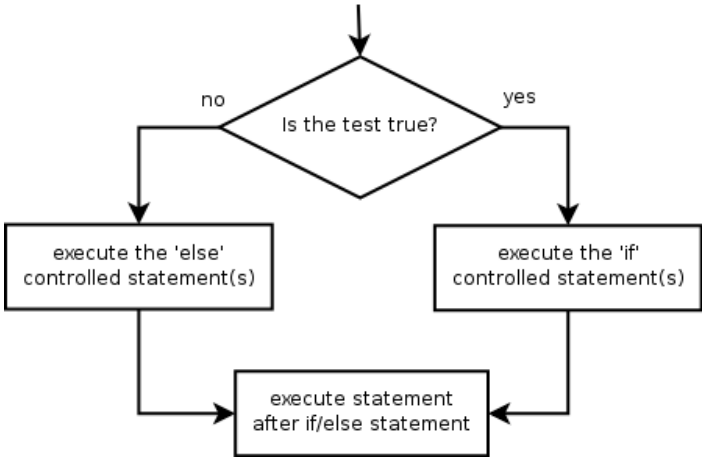
```
gpa = 3.4
if gpa > 2.0:
    print "Your application is accepted."
```



# 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
```

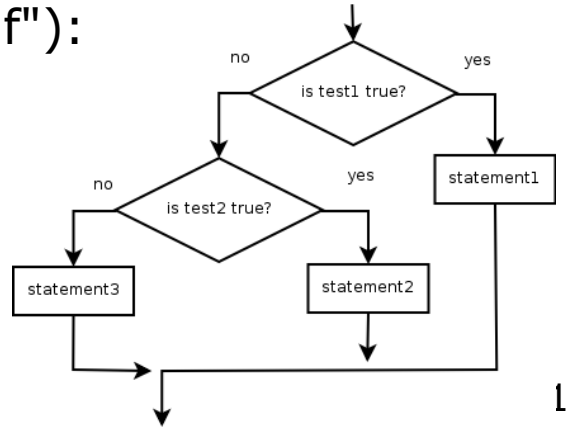



- Example:

```
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:

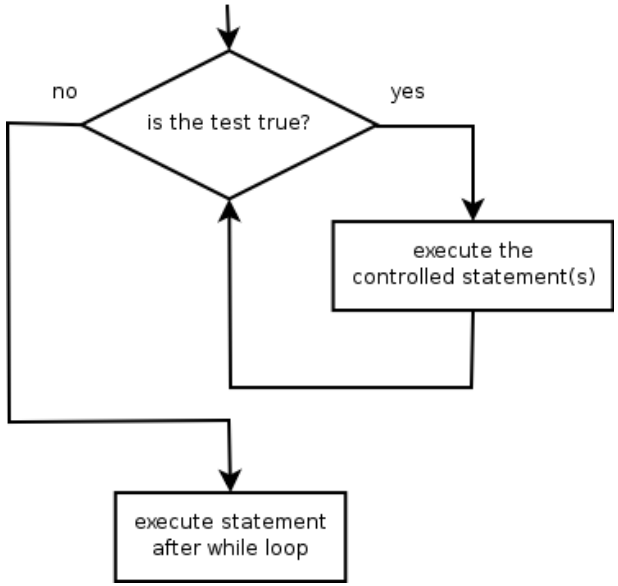
```
while condition:
    statements
```

- 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 | `1 + 1 == 2` | `True` |
| != | does not equal | `3.2 != 2.5` | `True` |
| < | less than | `10 < 5` | `False` |
| > | greater than | `10 > 5` | `True` |
| <= | less than or equal to | `126 <= 100` | `False` |
| >= | greater than or equal to | `5.0 >= 5.0` | `True` |

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

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

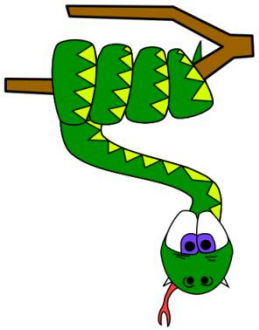
```
%python forloop1.py
1
7
13
2
```

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
1.2323200000000001
>>> 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'
```

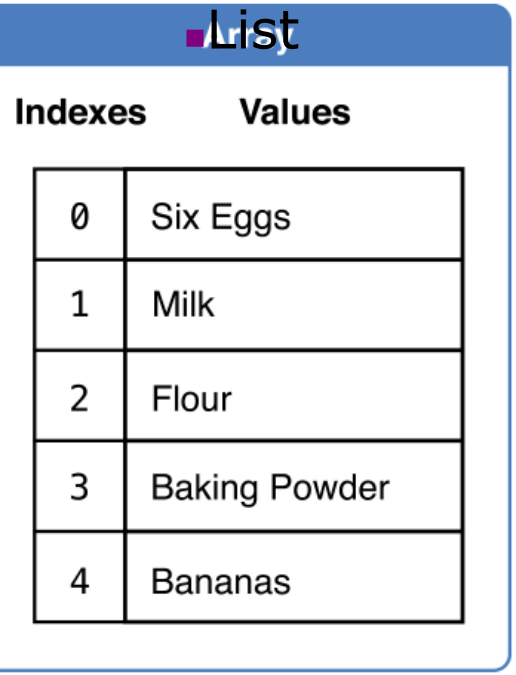
# String Formatting

- 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.

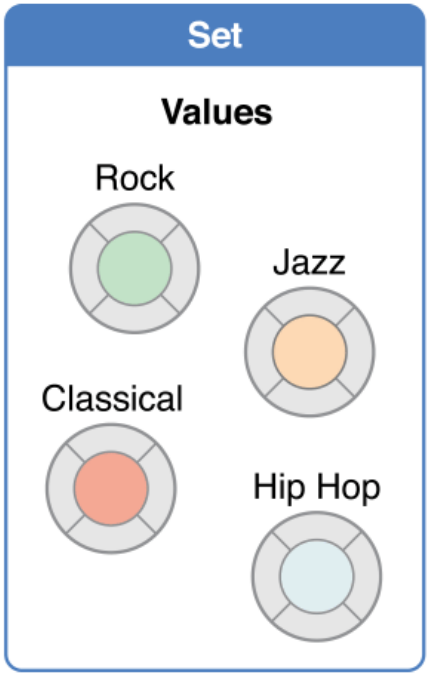
```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

# Types for Data Collection List, Set, and Dictionary

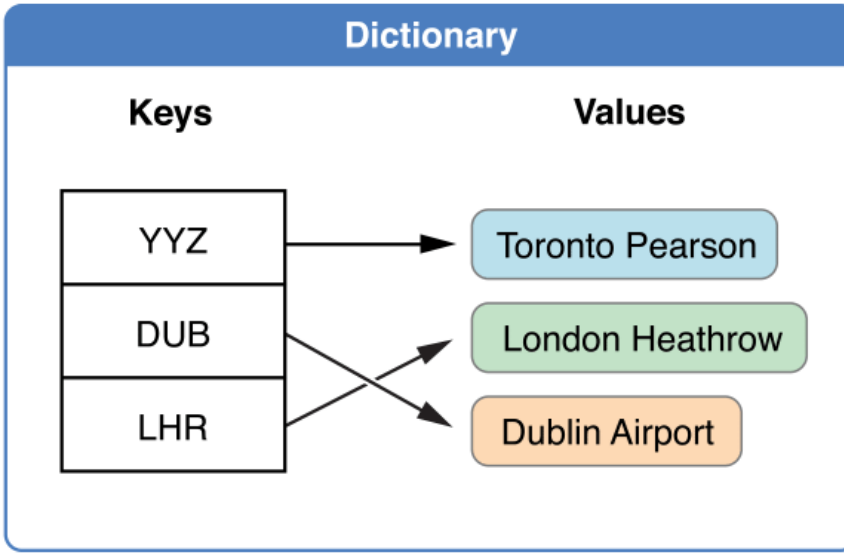
- List




- Ordered




- Unordered list




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 ith 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**

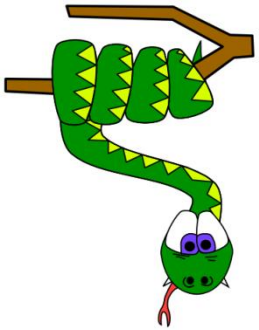
```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> 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




# Functions

# Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

# Functions are objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

# Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

# Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

funcasparam.py

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

- Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

# Higher-Order Functions

- **map(func,seq)** – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

# Higher-Order Functions

- **filter(boolfunc,seq)** – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

# Higher-Order Functions

- **reduce(func,seq)** – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

highorder.py

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

# Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

# Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
% python funcreturnfunc.py
<function bar at 0x612b0>
5
```

# Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

# Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

# Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

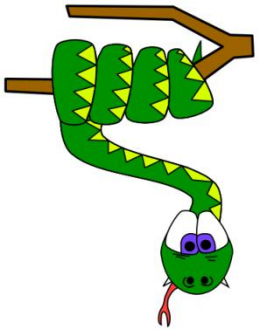
```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

# Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

# Modules: Imports

| | |
|---|---|
| import mymodule | Brings all elements of mymodule in, but must refer to as mymodule.<elem> |
| from mymodule import x | Imports x from mymodule right into this namespace |
| from mymodule import * | Imports all elements of mymodule into this namespace |




# Text and File Processing

# Strings

- **string**: A sequence of text characters in a program.
  - Strings start and end with quotation mark `"` or apostrophe `'` characters.
  - Examples:

```
"hello"
"This is a string"
"This, too, is a string.   It can be very long!"
```

- A string may not span across multiple lines or contain a " character.

```
"This is not
a legal String."
"This is not a "legal" String either."
```

- A string can represent characters by preceding them with a backslash.
  - `\t` tab character
  - `\n` new line character
  - `\"` quotation mark character
  - `\\` backslash character
  - Example: `"Hello\tthere\nHow are you?"`

# Indexes

- Characters in a string are numbered with *indexes* starting at 0:
  - Example:

```
name = "P. Diddy"
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| character | P | . |  | D | i | d | d | y |

- Accessing an individual character of a string:

***variableName*** `[` ***index*** `]`

  - Example:

```
print name, "starts with", name[0]
```

Output:

```
P. Diddy starts with P
```

# String properties

- `len(`***string***`)` - number of characters in a string (including spaces)
- `str.lower(`***string***`)` - lowercase version of a string
- `str.upper(`***string***`)` - uppercase version of a string

- Example:

```
name = "Martin Douglas Stepp"
length = len(name)
big_name = str.upper(name)
print big_name, "has", length, "characters"
```

Output:

```
MARTIN DOUGLAS STEPP has 20 characters
```

# raw_input

- `raw_input` : Reads a string of text from user input.
  - Example:

```
name = raw_input("Howdy, pardner. What's yer name? ")
print name, "... what a silly name!"
```

Output:

```
Howdy, pardner. What's yer name? Paris Hilton
Paris Hilton ... what a silly name!
```

# Text processing

- **text processing**: Examining, editing, formatting text.
  - often uses loops that examine the characters of a string one by one

- A `for` loop can examine each character in a string in sequence.
  - Example:

```
for c in "booyah":
    print c
```

Output:

```
b
o
o
y
a
h
```

# Strings and numbers

- `ord(`***text***`)` - converts a string into a number.
  - Example: `ord("a")` is `97`, `ord("b")` is `98`, ...
  - Characters map to numbers using standardized mappings such as *ASCII* and *Unicode*.
- `chr(`***number***`)` - converts a number into a string.
  - Example: `chr(99)` is `"c"`
- **Exercise:** Write a program that performs a rotation cypher.
  - e.g. `"Attack"` when rotated by 1 becomes `"buubdl"`

# File processing

- Many programs handle data, which often comes from files.

- Reading the entire contents of a file:

  ***variableName*** `= open("`***filename***`").read()`

  Example:

  ```
  file_text = open("bankaccount.txt").read()
  ```

# Line-by-line processing

- Reading a file line-by-line:

```
for line in open("filename").readlines():
    statements
```

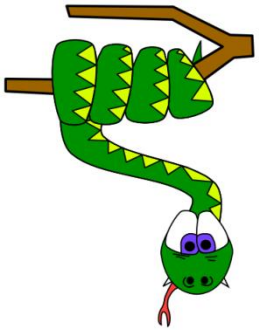
Example:

```
count = 0
for line in open("bankaccount.txt").readlines():
    count = count + 1
print "The file contains", count, "lines."
```

- **Exercise:** Write a program to process a file of DNA text, such as:

```
ATGCAATTGCTCGATTAG
```

  - Count the percent of C+G present in the DNA.




# Objects and Classes

# Defining a Class

- Python program may own many objects
  - An object is an item with fields supported by a set of method functions.
    - An object can have several fields (or called attribute variables) describing such an object
    - These fields can be accessed or modified by object methods
  - A class defines what objects look like and what functions can operate on these object.

- Declaring a class:

```
class name:
    statements
```

- Example:

```
class UCSBstudent:
   age = 21
    schoolname='UCSB'
```

# Fields

**name = value**

- Example:

```
class Point:
    x = 0
    y = 0

# main
p1 = Point()
p1.x = 2
p1.y = -5
```

**point.py**

```
1  class Point:
2      x = 0
3      y = 0
```

- can be declared directly inside class (as shown here) or in constructors (more common)
- Python does not really have encapsulation or private fields
  - relies on caller to "be nice" and not mess with objects' contents

# Using a Class

`import` **class**

- client programs must import the classes they use

**point_main.py**

```
1  from Point import *
2
3  # main
4  p1 = Point()
5  p1.x = 7
6  p1.y = -3
7
8  p2 = Point()
9  p2.x = 7
10 p2.y = 1

   # Python objects are dynamic (can add fields any time!)
   p1.name = "Tyler Durden"
```

# Object Methods

```
def name(self, parameter, ..., parameter):
    statements
```

- `self` *must* be the first parameter to any object method
  - represents the "implicit parameter" (`this` in Java)
- *must* access the object's fields through the `self` reference

```
class Point:
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

# Exercise Answer

**point.py**

```
1  from math import *
2
3  class Point:
4      x = 0
5      y = 0
6
7      def set_location(self, x, y):
8          self.x = x
9          self.y = y
10
11     def distance_from_origin(self):
12         return sqrt(self.x * self.x + self.y * self.y)
13
14     def distance(self, other):
15         dx = self.x - other.x
16         dy = self.y - other.y
17         return sqrt(dx * dx + dy * dy)
```

# Calling Methods

- A client can call the methods of an object in two ways:
  - (the value of `self` can be an implicit or explicit parameter)

1) **object.method(parameters)**

or

2) **Class.method(object, parameters)**

- Example:

```
p = Point(3, -4)
p.move(1, 5)
Point.move(p, 1, 5)
```

# Constructors

```
def __init__(self, parameter, ..., parameter):
    statements
```

- a constructor is a special method with the name `__init__`
- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    ...
```

  - How would we make it possible to construct a `Point()` with no parameters to get (0, 0)?

# toString and __str__

```
def __str__(self):
    return string
```

- equivalent to Java's `toString` (converts object to a string)
- invoked automatically when `str` or `print` is called

Exercise: Write a `__str__` method for `Point` objects that returns strings like `"(3, -14)"`

```
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

# Complete Point Class

**point.py**

```
 1  from math import *
 2
 3  class Point:
 4      def __init__(self, x, y):
 5          self.x = x
 6          self.y = y
 7
 8      def distance_from_origin(self):
 9          return sqrt(self.x * self.x + self.y * self.y)
10
11      def distance(self, other):
12          dx = self.x - other.x
13          dy = self.y - other.y
14          return sqrt(dx * dx + dy * dy)
15
16      def move(self, dx, dy):
17          self.x += dx
18          self.y += dy
19
20      def __str__(self):
21          return "(" + str(self.x) + ", " + str(self.y) + ")"
```

# Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

| Operator | Class Method |
|---|---|
| - | `__neg__(self, other)` |
| + | `__pos__(self, other)` |
| * | `__mul__(self, other)` |
| / | `__truediv__(self, other)` |

- Unary Operators

| | |
|---|---|
| - | `__neg__(self)` |
| + | `__pos__(self)` |

| Operator | Class Method |
|---|---|
| == | `__eq__(self, other)` |
| != | `__ne__(self, other)` |
| < | `__lt__(self, other)` |
| > | `__gt__(self, other)` |
| <= | `__le__(self, other)` |
| >= | `__ge__(self, other)` |

# Generating Exceptions

`raise` **`ExceptionType`**`("`**`message`**`")`

- useful when the client uses your object improperly
- types: `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`

- Example:

```
class BankAccount:
    ...
    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative amount")
        ...
```

# Inheritance

```
class name(superclass):
    statements
```

- Example:

```
class Point3D(Point):    # Point3D extends Point
    z = 0
    ...
```

- Python also supports *multiple inheritance*

```
class name(superclass, ..., superclass):
    statements
```

*(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)*

# Calling Superclass Methods

- methods: **class.method(object, parameters)**
- constructors: **class.__init__(parameters)**

```
class Point3D(Point):
    z = 0
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def move(self, dx, dy, dz):
        Point.move(self, dx, dy)
        self.z += dz
```