# CS 457

Module 2 Python

# Agenda

- Administration
  - PHP Midterm announcements
- Python
- Assignment 1

# Administration

- PHP midterm 1
  - Some questions have a formatting issue, code does not show up
  - Those questions will be given fullpoints.
  - If you have any issues during the exam, take a screen shot and send me the documentation.
  - Test will be graded soon

# Administration

- Assignment 1

  I am preparing a git repository for you to turn in your code. Currently we are testing the server. The server should be available next class.

# Module 2 – Python

- The next scripting language we will cover is **Python**

# Python

- Python is a high-level, interpreted, interactive and object-oriented scripting language.
  - **Interpreted** – processed at runtime by the interpreter no need to compile your program before running
  - Object-Oriented – encapsulate code within objects
  - Interactive – has a shell propt you can interact with
- Python has an elegant syntax, with many natural language features

# Python: History

- Developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Available under the GNU General Public License (GPL).

# Python: which version?

- There are several versions of python

  Version 2.x Versus 3.x

  Which is better and why?

  Same with only minor differences

  We will be using 3.x latest stable release.

# Python: which version?

We will be using 3.x latest stable release.

Justification,

“*Python 2.x is legacy, Python 3.x is the present and future of the language*”

# Python: Distributions

The official version

www.python.org

A really good scientific version

Anaconda from Continuum Analytics

# Python: Distributions

Which to choose?

I prefer the anaconda version, you may use either one.

Both have package managers to add functionality and distributions to your install.

# Python IDEs

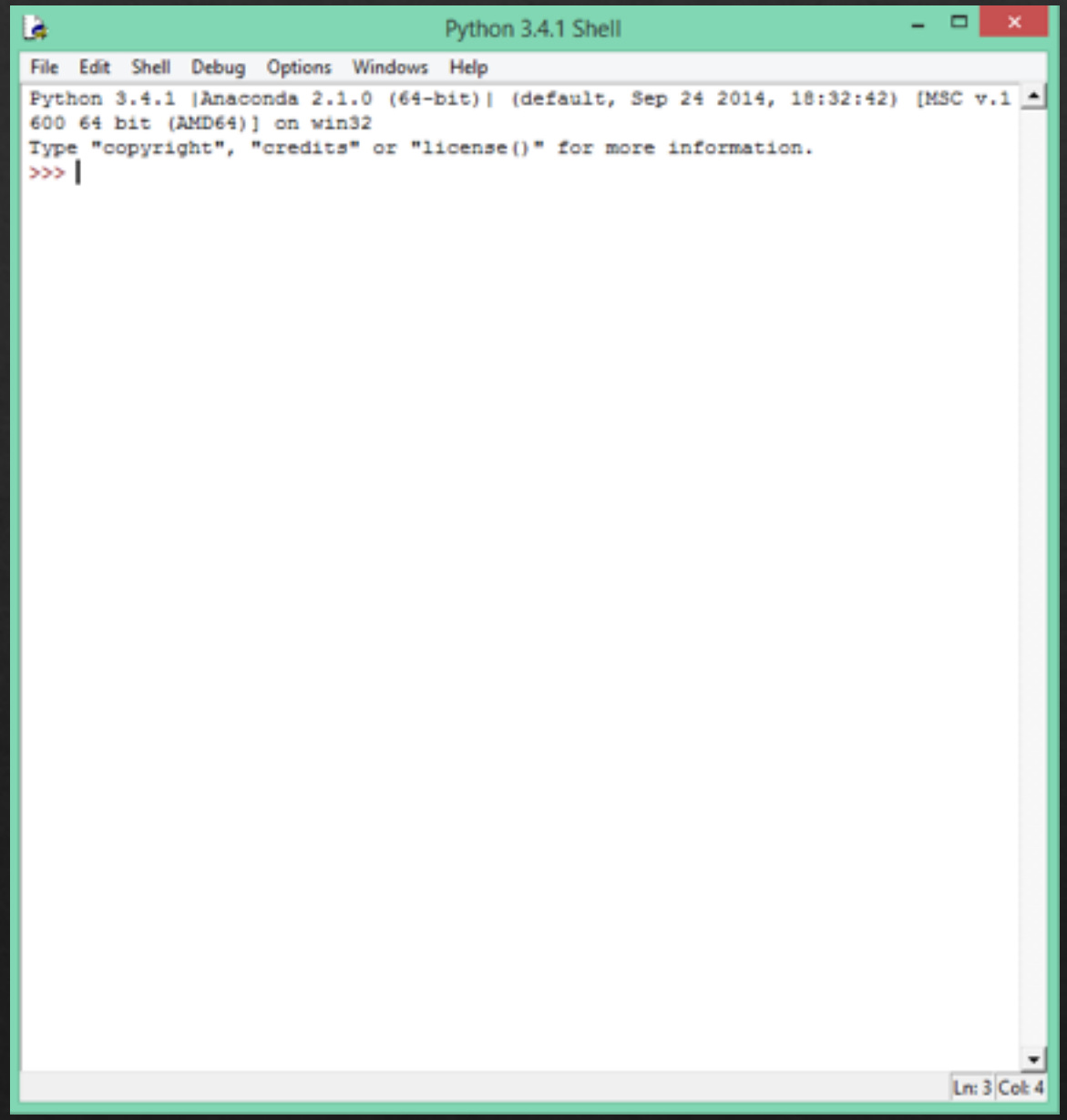
- Since python is an interpreted language, an IDE is not needed.
- The reality is that a ***good*** IDE is **highly recommended**

# Python IDEs

- Built in – IDLE
- Pycharm from JetBrains
- Pydev for Eclipse

# Python IDEs

- Built in – IDLE

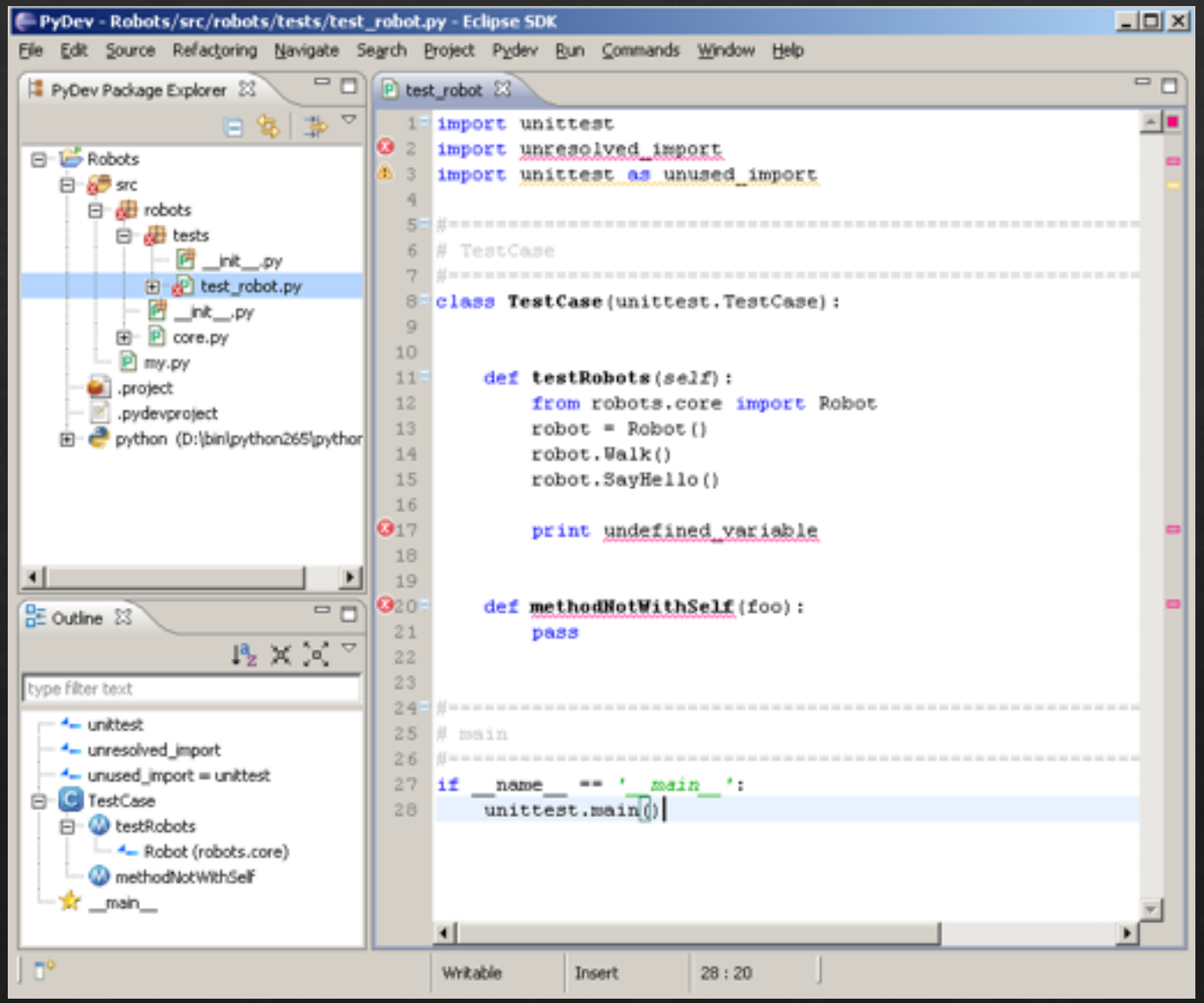


# Python IDEs

- Pydev for Eclipse

**PyDev's Official Site:**

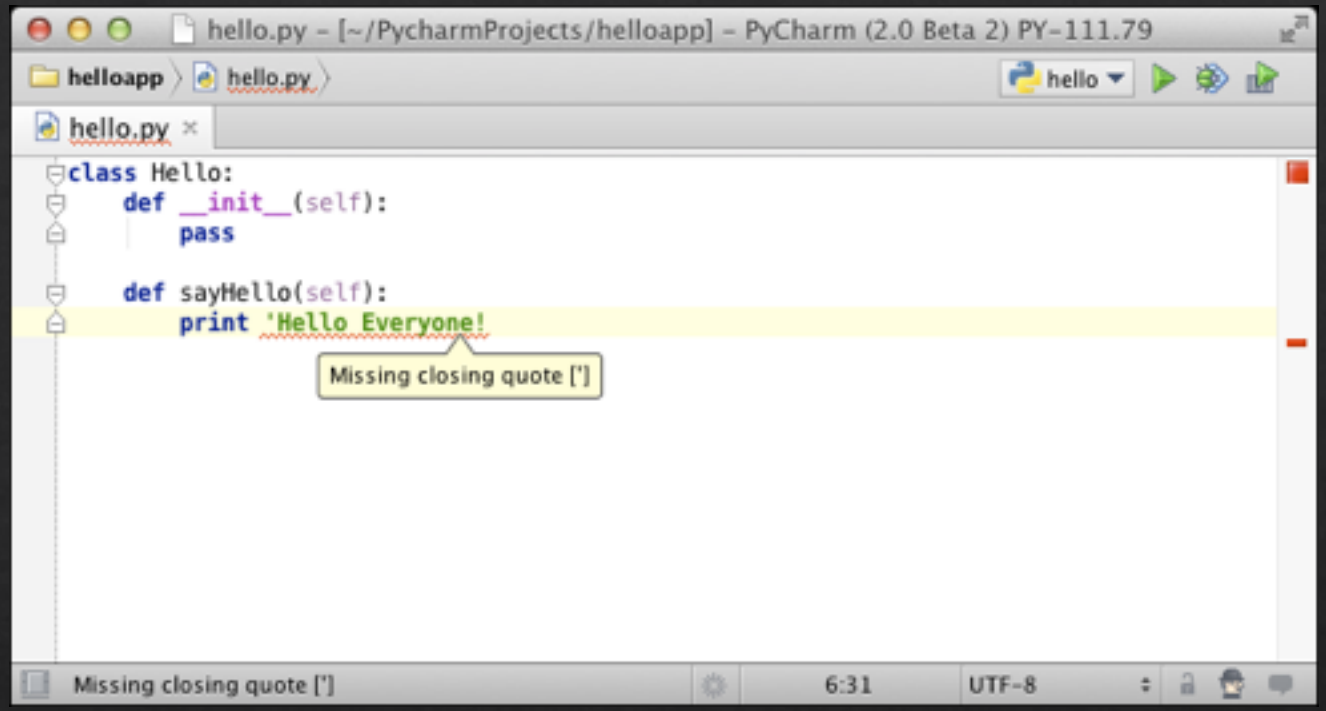
**http://pydev.org/**



# Python IDEs

- Pycharm from JetBrains

**PyCharm's Official Site:**




http://www.jetbrains.com/pycharm/

# Demo and Assignment

- Download Python
- Download an IDE(s) try one
- Run the following program

# Demo and Assignment

Assignment: set up your python development environment and run the following code the calculate the fibonaci series. Bonus, make it take user input.

```
def fib(n):
    a, b= 0, 1
    while a < n:
        print (a, end=' ')
        a, b = b, a+b
    print()

fib(1000)
```

# CS 457

Python Language Basics

Week 8 Day 2

# Agenda

- Administration
  - PHP Midterm announcements
- Python Language
- Assignment 1

# Administration

- PHP midterm 1
  - I could not reset the test individually, so I had to globally rese the exam.
  - Two attempt please
  - Test will be graded this weekend

# Administration

- Assignment 1
  - The server has been delayed
  - The server should be available next class.

# Python

- In todays class we will cover some of the language elements of the python language.

- Before that we will review from last class.

# Python

- Python is a high-level, interpreted, interactive and object-oriented scripting language.
- Python has an elegant syntax, with many natural language features

# Python: which version?

We will be using 3.x latest stable release.

Justification,

“*Python 2.x is legacy, Python 3.x is the present and future of the language*”

# Python IDEs

- Does anyone have any input?

# Python IDEs

- Built in – IDLE
- Pycharm from JetBrains
- Pydev for Eclipse

# Python Language Elements

- Arithmetic
- Strings
- and Variables

# Python: Basic Data Types

- Numbers
  - Integers ( whole numbers)
  - Floating Point  ( 12.345)
- Strings

# Python: Advanced Data Types

- Complex numbers
- Sequences
- Tuples
- Lists
- List Functions
- Dictionaries
- Sets
- and more

# Python: Numbers

Integers

```
>>> 3
3
>>> 3 + 3
6
```

# Python: Numbers

Integers

>>> 6 / 2

3.0

>>> 6 // 2

3

Note the // operator is the Integer division operator. It always returns an integer. The digits after the decimal are chopped off. No rounding

>>> 6//5

1

# Python: Integers

Integers are unlimited in size, unlike other programming languages.

```
>>> 273 ** 100
```

```
4132993326918459867994966473521663173302453387280390885146144614327342052420820943773286264317154220472376275701312952356825556540721745114150356802019042527361095511590090238997885948781930095832624201478477489459713702233424449935136235176001
```

# Python: Floating point arithmetic

Floating point numbers:

-3.14159

Large numbers are qriten with scientific notation

>>> 3.14159e0

3.14159

No leading 0 required

>>>.5

0.5

# Python: Floating point arithmetic

Unlike integers, floats have a minimum and maximum values that if exceeded will cause overflow errors.

These can be silent errors that are difficult to debug

```
>>> 500.0 **10000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
>>>
```

# Python: Floating point arithmetic

Precision – limited by the digital nature of comuters

Consider:

```
>>> 1 - 2 / 3
0.33333333333333337
```

Should have an infinite number of 3s (not a 7)

# Python: Complex arithmetic

- Complex numbers are useful in certain engineering and scientific calculations.

Complex numbers involve the $\sqrt{-1}$

In python **1j** denotes $\sqrt{-1}$

```
>>> 1j
1j
>>>
>>> 1j * 1j
(-1+0j)
```

# Python: Useful tip

Python is mostly a self-documenting language.

Most functions and modules come with brief explanations to help you figure out how to use them

To list functions within a module use

```
dir(moduleName)
```

# Python: Other common math functions

Python has a bunch of built in mathematic functions. To access them you need to import the math module. At you python prompt try these commands:

```
>>>import math
>>>dir(math)
>>>print(math.__doc__)
>>>print(math.cos.__doc__)
```

# Python: Other common math functions

```
>>> print (math.__doc__)
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
>>>
>>> print (math.cos.__doc__)
cos(x)

Return the cosine of x (measured in radians).
```

# Python: Strings

Strings are a secuence of one or more characters (Unicode) such as

“catdog”

“619-594-1710”

“Python is fun!”

# Python: Strings

Python has three ways of indicating string literals:

- Single quotes such as ‘http’ or ‘Hello’, or ‘CATDOG’
- Double quotes such as “http”, or “Helo” or “CATDOG”
- Triple quotes such as “”””http””””  or multiline strings such as

  ,,,,,,

  Open up a book

  and start reading

  ,,,,,,

# Python: Strings

Triple quotes can use ’’’ the single quote character.

Single and double quote strings can also contain single and double quotes inside them

“It’s great!”

or

‘She said “He went to Jarods!” with great enthusiasim‘

# Python: Strings

To determine the number of characters in a string use the len(string) function:

```
>>>len('python')
6
```

# Python: Strings

To concatenate  two strings user the +

>>>’cat’ + ‘dog’

‘catdog’

Use the * to concatenate many times

>>>’cat’ + ‘dog’ * 3

‘catdogdogdog’

# Python – Type conversion

Python provides a number of built in functions to convert from one type to another.

```
>>>float('3.2')
3.2
>>>float(3)
3.0
```

# Python – Type conversion

Sometime python will convert between numeric types automatically.

This is known as implicit conversions

```
>>> 25 * 8.5
212.5
```

# Conversion – float to an integer

The int(x) function is used to convert to an integer. but care must be used

```
>>> int(8.64)
8
```

# Conversion – float to an integer

The round(x) function can also be used to convert to an integer. However be aware of the following quirks

```
>>> round(8.64)
9
>>> round(8.5)
8
```

# Conversion – strings to numbers

behave as you would expect

```
>>> int('5')
5
>>> float('98.6')
98.6
```

# Variables and values

In python variables ***label***, or ***point to***, a value.

```
>>> fruit = 'apple'
>>> fruit
'apple'
```

# Variables Names

- A variable name can be of any length, although the characters in it must be either letters, numbers, or the under-score character (_). Spaces, dashes, punctuation, quotation marks, and other such characters are not allowed
- The first character of a variable name cannot be a number, it must be a letter or an underscore (_) character
- Python is case sensitive.Thus CAT, Cat, and cat are three different variables.
- Keywords are not allowed as variable names

# Assignment Statements

- Assignment statements have three main parts:
  1. left-hand side
  2. the assignment operator
  3. the right-hand side

```
>>>  var = value
```

# Assignment Statements

- Assignment statements don’t copy. What this means is that assignment statements don’t make a copy of the value they point to. All they do is label and relabel existing values.

```
>>>rate =0.04
>>>rate_2008=0.06
>>> rate = rate_2008
>>>rate
0.06
```

# Assignment Important feature

- Strings and Numbers are ***immutable*** in python.

  They cannot be changed in any way ever. Whenever it seams like you are changing a number or a string you are really changing a copy of the  variable

```
>>> s='apple'
>>> s+'s'
'apples'
>>> s
'apple'
>>> 5 = 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

# Multiple Assignment

- Python has a convenient trick that lets you assign more than one variable at a time.

```
>>> x, y, z = 1, 'two', 3.0
>>> x
1
>>> y
'two'
>>> z
3.0
```

# Multiple Assignment

- This can be used for swapping in parallel with out a temp variable

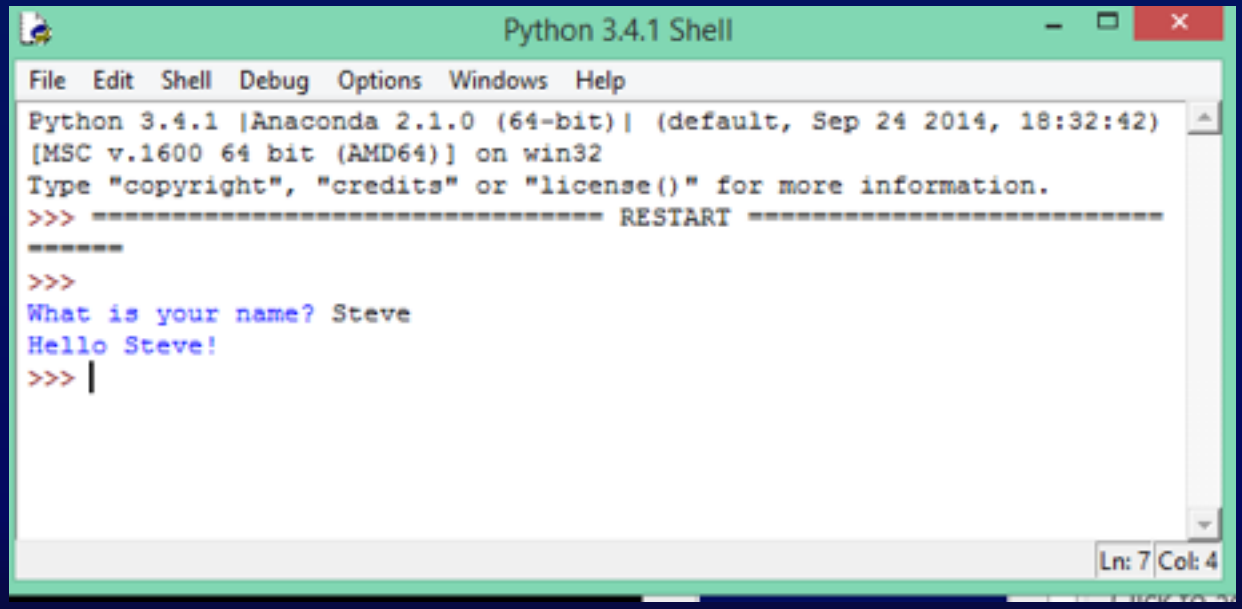
```
>>> a, b = 5, 9
>>> a,b
(5, 9)
>>> a,b = b, a
>>> a, b
(9, 5)
```

# reading Strings from the keyboard

Example: open up idle and type in the following code

```
name = input('What is your name? ')
print ('Hello ' + name.capitalize() + '!')
```

Then save and run the program.

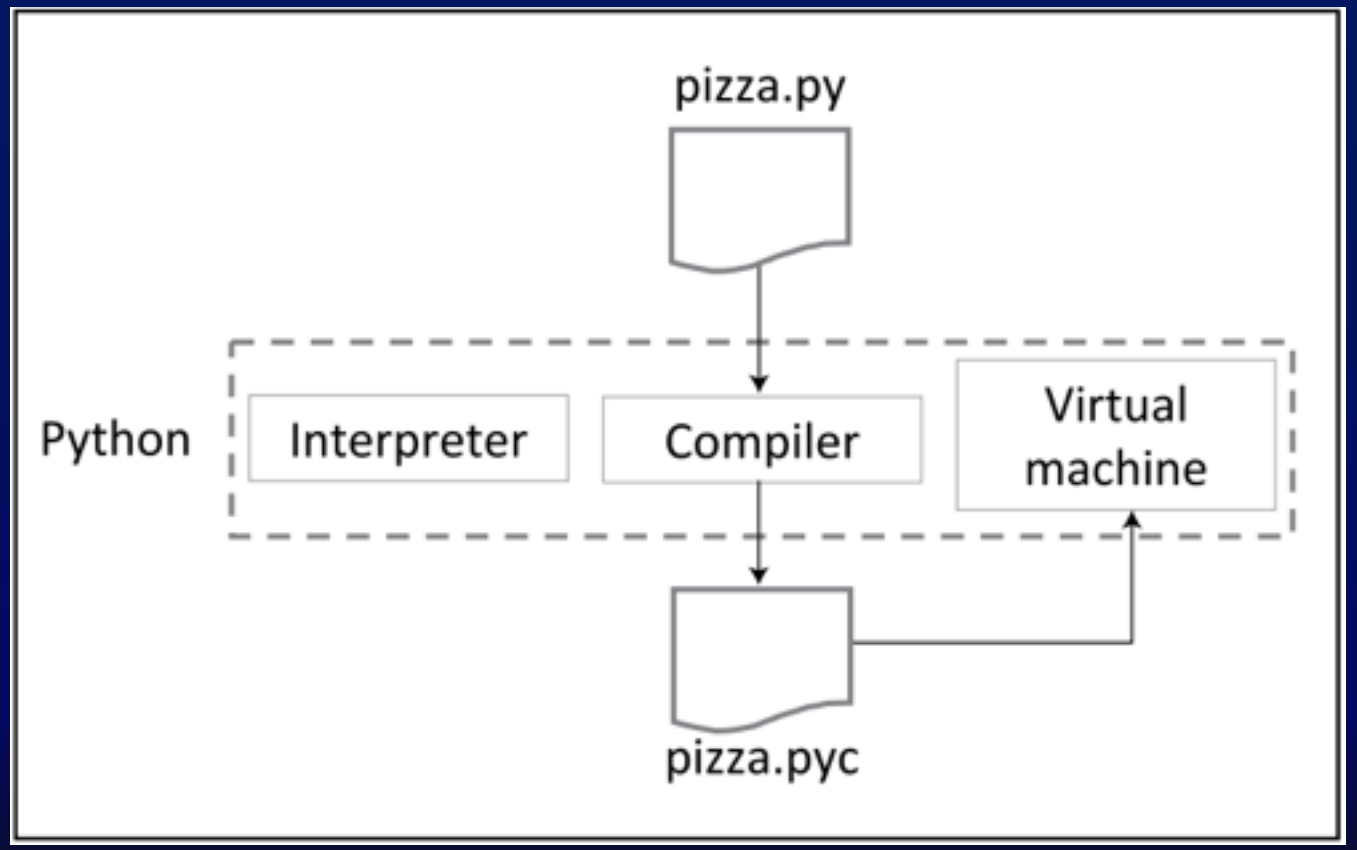


# Compiling Source code

in the previous example python is converting your python code to object code. This process consists of three parts:

1) the Interpreter
2) the Compiler
3) Virtual Machine

The virtual machine is the reason why your pyc code can run on different computers without modification.



# Printing Strings to the screen

The print function is what is used to display results to the user.

```
name = input('What is your name? ')
print ('Hello ' + name.capitalize() + '!')
```

# Comments

Python uses the # symbol to signify a source code comment

```
# get the users name
name = input('What is your name? ')
#
print ('Hello ' + name.capitalize() + '!')  # print it to the screen
```

# Flow Control

python lets you change the flow of the program with if/else – statements. If statements have the form:

```
if (Boolean expression):
    statement1
    statement2
else:
    statement3 in else
statement not in block
```

# Loops in python

There are two main types of loops in python

- for-loops
- while-loops

# For loop

```
for i in range(1, 11):
    print (i + 1)
```

```
>>>
2
3
4
5
6
7
8
9
10
11
```

# While loop

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

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

# Loops homework

Write two simple python programs to calculate the factorial of a number using both a for loop and a while loop.

Hint 6! = 1 x 2 x 3 x 4 x 5 x 6

and 1! = 1

# Assignment 1

- Any questions?

# CS 457

Python Language Continuation

Week 9 Day 1

3/17/2015

# Agenda

- Administration
  - None
- Python Language Continues
- Assignment 1

# Administration

# Python

- In todays class we will continue to cover the language elements of the python language.

- Before that we will review from last class.

# Loops homework

Write two simple python programs to calculate the factorial of a number using both a for loop and a while loop.

Hint 6! = 1 x 2 x 3 x 4 x 5 x 6

and 1! = 1

# Loops in python

There are two main types of loops in python

- for-loops
- while-loops

# For loop

```
for i in range(1, 11):
    print (i + 1)
```

```
>>>
2
3
4
5
6
7
8
9
10
11
```

# Range function

The range() function is used to generate a series of numbers (iterables) in python. The format is:

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

Start and stop are optional parameters.

If start is omitted, it defaults to 0;

If stop is omitted, it defaults to 1;

If step is positive, the last element is the largest start + i * step less than stop;

If step is negative, the last element is the smallest start + i * step greater than stop. step must not be zero (or else ValueError is raised).

# Factorial using For Loops in Python

Open up your python ide or idle and try

```
n = int(input("Enter an integer greater than 0: " ))
fact = 1
for i in range(2, n + 1):
    fact = fact * i
print (str(n) + ' factorial is ' + str(fact))
```

# While loop

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

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

# Factorial using For Loops in Python

Open up your python ide or idle and try

```
n = int(input("Enter an integer greater than 0: " ))
fact = 1
i = 1
while i < n+1:
    fact = ???? * i
    i= i + 1
print (str(n) + ' factorial is ' + str(fact))
```

# Python: Language types

| | Object type | Example literals/creation |
|---|---|---|
| √ | Numbers | 1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction() |
| √ | Strings | 'spam', "Bob's", b'a\x01c', u'sp\xc4m' |
| | Lists | [1, [2, 'three'], 4.5], list(range(10)) |
| | Dictionaries | {'food': 'spam', 'taste': 'yum'}, dict(hours=10) |
| | Tuples | (1, 'spam', 4, 'U'), tuple('spam'), namedtuple |
| | Files | open('eggs.txt'), open(r'C:\ham.bin', 'wb') |
| | Sets | set('abc'), {'a', 'b', 'c'} |
| | Other core types | Booleans, types, None |
| | Program unit types | Functions, modules, classes |

# Lists Types

Lists are

- Positionally ordered collections of arbitrarily typed objects
- Have no fixed size.
- are mutable – can be changed in plase
- Use [ ] notation to denote

Example

```
>>> L = [123, ‘AbC', 1.23]          # A list of three different-type objects
>>> len(L)                              # Number of items in the list
3
```

# Lists Types

With list one can index, slice, and so on, just as for strings:

```
>>> L[0]                        # Indexing by position
123
>>> L[:-1]                      # Slicing a list returns a new list
[123, ‘AbC']

>>> L + [4, 5, 6]               # Concat/repeat make new lists too
[123, ‘AbC', 1.23, 4, 5, 6]
>>> L * 2
[123, ' AbC ', 1.23, 123, ' AbC ', 1.23]

>>> L                           # We're not changing the original list
[123, ' AbC ', 1.23]
```

# Lists – type specific operations

Further, lists have no fixed size. That is, they can grow and shrink on demand, in response to list-specific operations:

```
>>> L.append(‘In')                  # Growing: add object at end of list
>>> L
[123, ‘AbC', 1.23, ‘In']

>>> L.pop(2)                        # Shrinking: delete an item in the middle
1.23
>>> L                               # "del L[2]" deletes from a list too
[123, ‘AbC', 'In']
```

# Lists – type specific operations

Other list methods insert an item at an arbitrary position (insert), remove a given item by value (remove), add multiple items at the end (extend), and so on.

L.insert(2, "foo")

[123, 'AbC', 'foo', 1.23]

# Lists – type specific operations

Because lists are mutable, most list methods also change the list object in place, instead of creating a new one:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

# Lists – Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end:

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range

>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

# Lists – Nesting

Pythons supports arbitrary nesting, as many levels and  in any combination. With this we can create ‘multi-dimensional’ arrays.

```
M = [[1, 2, 3],     # 3 x 3
     [4, 5, 6],     # multi lines is Bracketed
     [7, 8, 9]]

print (M)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

# Lists – Accessing Nested elements

```
M = [[1, 2, 3],     # 3 x 3
     [4, 5, 6],     # multi lines is Bracketed
     [7, 8, 9]]

>>> M[1]                    # Get row 2
[4, 5, 6]

>>> M[1][2]                  # Get row 2, then get item 3 within the row
6
```

# Comprehensions

What is we wanted to get the second column of our array M?

```
M = [[1, 2, 3],
     [4, 5, 6],     # multi lines is Bracketed
     [7, 8, 9]]
```

In python we would use a list comprehension method like such:

```
col2 = [row[1] for row in M]
```

**[2, 5, 8]**

# Comprehensions

**List comprehensions derive from set notation**
**they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right.**
**List comprehensions are coded in square brackets [] (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name (row, here).**
**They are quite powerful and can get complex**

# Comprehensions

```
>>> [row[1] + 1 for row in M]                  # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]   # Filter out odd items
[2, 8]

col2 = [[row[0], row[2]] for row in M]
[[1, 3], [4, 6], [7, 9]]
```

# Comprehensions - exercise

What would you use to find the diagonal of M?

```
M = [[1, 2, 3
     [4, 5, 6],
     [7, 8, 9]]
```

# Comprehensions – and other data types

Comprehension can be used to create ***sets*** and ***dictionaries*** too.

```
>>> {sum(row) for row in M}              # Create a set of row sums
{24, 6, 15}

>>> {i : sum(M[i]) for i in range(3)}    # Creates key/value table of row sums
{0: 6, 1: 15, 2: 24}
```

# Python – Dictionaries

In python dictionaries are not sequences, but rather ***mappings*** instead. For example:

**D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}**

**Dictionaries**

- Store values based on a key instead of position
- Almost always right to left, but not necessarity always in the same order
- Are mutable
- well suited for mnemonic keys – labeled keys

# Python – Dictionaries

```
>>> D['food']              # Fetch value of key 'food'
'Spam'

>>> D['quantity'] += 1     # Add 1 to 'quantity' value
>>> D
          {'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

# Dictionaries creation

```
>>> D = {}
>>> D['name'] = 'Bob'      # Create keys by assignment
>>> D['job']  = 'dev‘       # no out of bounds issues like with lists.
>>> D['age']  = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}
>>> print(D['name'])
Bob
```

# Dictionaries creation by keywords

In python one can make dictionaries by passing to the **dict** type name keyword arguments (a special name=value syntax in function calls)

```
>>> bob1 = dict(name='Bob', job='dev', age=40)                # Keywords
>>> bob1
{'age': 40, 'name': 'Bob', 'job': 'dev'}
```

# Dictionaries creation by zipping

Dictionaries can also be created as the result of ***zipping*** together sequences of keys and values obtained at runtime (e.g., from files)

```
>>> bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40]))    # Zipping
>>> bob2
{'job': 'dev', 'name': 'Bob', 'age': 40}
```

# Dictionary Nesting[nesting[nesting[nesting[nesting]]]]

In Python complex data can be represented via nesting dictionaries. Suppose in the previous example we wanted to represent a person with first and last names and multiple jobs?

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
    'jobs': ['dev', 'mgr'],
    'age':  40.5}
```

# Dictionary Nesting[nesting[nesting[nesting[nesting]]]]

Accessing is done by key:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
    'jobs': ['dev', 'mgr'],
    'age':  40.5}
>>> rec['name']                    # 'name' is a nested dictionary
    {'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']            # Index the nested dictionary
    'Smith'

    >>> rec['jobs']               # 'jobs' is a nested list
    ['dev', 'mgr']
>>> rec['jobs'][-1]               # Index the nested list
    'mgr'
```

# Dictionary Nesting[nesting[nesting[nesting[nesting]]]]

Information can easily be apprended to the record:

```
>>> rec['jobs'].append('janitor')       # Expand Bob's job description in place
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith', 'first': 'Bob'}}
```

# Dictionaries and Missing keys

- In python assigning a non existing key grows the disctionary; but referencing an non existing key is an error. Consider,

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                    # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                         # Referencing a nonexistent key is an error
...error …KeyError: 'f'
```

# Dictionaries and Missing keys

- Question: But what if we don't know the keys in advance?

  Answer: Test with the **in** dictionary membership function

```
>>> 'f' in D
False

>>> if not 'f' in D:
print('missing')

missing
```

# Dictionaries Sorting keys

Because dictionaries are not sequences, they don't maintain any dependable left-to-right order. If we make a dictionary and print it back, its keys may come back in a different order than that in which we typed them, and may vary per Python version and other variables:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

So how do we reliably sort them?

# Dictionaries Sorting keys

One way is to create a list of keys and then sort the key list

```
Ks = list(D.keys())
Ks.sort()
for key in Ks:                    # Iterate though sorted keys
        print(key, '=>', D[key])
```

OR in 3.x use

```
for key in sorted(D):
        print(key, '=>', D[key])
```

# Tuples

In python tuples are *sequences*, like lists, but they are *immutable*, like strings.

Functionally, they're used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations:

```
>>> T = (1, 2, 3, 4)          # A 4-item tuple
>>> len(T)                    # Length
4
```

# Tuples

Tuples also have type-specific callable methods as of Python 2.6 and 3.0, but not nearly as many as lists:

```
>>> T.index(4)              # Tuple methods: 4 appears at offset 3
3
>>> T.count(4)              # 4 appears once
1
```

# Tuples

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences (one-item tuples like the one here require a trailing comma):

```
>>> T[0] = 2                  # Tuples are immutable
...error
>>> T = (2,) + T[1:]          # Make a new tuple for a new value
>>> T
(2, 2, 3, 4)
```

# So Why Tuples?

So, why have a type that is like a list, but supports fewer operations? Frankly, tuples are not generally used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in larger 1programs.

# CS 457

Python Language Continuation

Week 9 Day 2

3/19/2015

# Agenda

- Administration
  - None
- Python Language
  - Files, Sets, Other types
- Functions
- Simple Programs

# Administration

# Python: Language types

| | Object type | Example literals/creation |
|---|---|---|
| √ | Numbers | 1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction() |
| √ | Strings | 'spam', "Bob's", b'a\x01c', u'sp\xc4m' |
| √ | Lists | [1, [2, 'three'], 4.5], list(range(10)) |
| √ | Dictionaries | {'food': 'spam', 'taste': 'yum'}, dict(hours=10) |
| √ | Tuples | (1, 'spam', 4, 'U'), tuple('spam'), namedtuple |
| | Files | open('eggs.txt'), open(r'C:\ham.bin', 'wb') |
| | Sets | set('abc'), {'a', 'b', 'c'} |
| | Other core types | Booleans, types, None |
| | Program unit types | Functions, modules, classes |

# Files

File objects are the main interface to external files on a computer for Python.

Can be any ‘type’ of file, e.g. text, binary, images, xml, media, web, etc

# Files

Use the open function to work with files:

commonly used with two parameters.

open(filename, mode)

filename = the name of the file

mode = an optional string that specifies the mode in which the file is opened.

# Files – modes

| Character | Meaning |
| --- | --- |
| 'r' | open for reading (default) |
| 'w' | open for writing, truncating the file first |
| 'x' | open for exclusive creation, failing if the file already exists |
| 'a' | open for writing, appending to the end of the file if it exists |
| 'b' | binary mode |
| 't' | text mode (default) |
| '+' | open a disk file for updating (reading and writing) |
| 'U' | universal newlines mode (deprecated) |

# Files – modes

The default mode is 'r' (open for reading text, synonym of 'rt').

For binary read-write access,

'w+b' opens and truncates the file to 0 bytes.

'r+b' opens the file without truncation.

# Files – Binary vs Text

Python distinguishes between binary and text I/O.
Files opened in binary mode (including 'b' in the mode argument) return contents as bytes objects without any decoding.

In text mode (the default, or when 't' is included in the mode argument), the contents of the file are returned as str, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

# Files – Open

The full signature of the open function (from the python maual is)

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,
        opener=None)¶
```

Python allows for specific behavior for *buffering*, *encoding*, *error* and *new line* handling and more.

# Files – Open

There are also the file handling modules, such as, fileinput, io (where open() is declared), os, os.path, tempfile, and shutil.

read more at https://docs.python.org/3/library/functions.html#open

# Files – Processing

There are some simple built in functions for reading information from a file:

read() Returns the entire remaining contents of the file as a single (potentially large, multi-line) string.

readline() Returns the next line of the file. That is all text up to and including the next newline character.

readlines() Returns a list of the remaining lines in the file. Each list item is a single line including the newline character at the end.

# Files – Example writing

```
>>> f = open('data.txt', 'w')
>>> f.write('Hello CS547\n')
12
>>> f.write('Python is Fun')
13
>>> f.close()
>>>
```

# Files – Example Reading

```
>>> f = open('data.txt')
>>> text = f.read()
>>> text
'Hello CS547\nPython is Fun'
>>> print(text)
Hello CS547
Python is Fun
```

# Files – Reading with Iterator

There is more than one way of reading a file ( seek, readline, and more) in python.

One convenient way is to use an iterator to automatically read the file line by line.

```
>>> for line in open('data.txt'): print('line = ' + line)
...
line = Hello CS547

line = Python is Fun
```

# Files – Note about text files

Be aware that not all text files are alike. Files from systems with other Unicode character set may not ‘look’ the same on your system. Especially if the two computers are in different countries.
Luckily, python makes Unicode interoperability easy.

To access files containing non-ASCII Unicode text, we simply pass in an encoding name if the text in the file doesn’t match the default encoding for our platform. In this mode, Python text files automatically encode on writes and decode on reads per the encoding scheme name you provide.

# Files – Unicode encoding text files

In Python 3.X:

```
S = 'sp\xc4m'
>>> S
'spÄm‘

>>> file = open('unidata.txt', 'w', encoding='utf-8')      # Write/encode UTF-8 text
>>> file.write(S)
4
>>> file.close()

>>> text = open('unidata.txt', encoding='utf-8').read()    # Read/decode UTF-8 text
>>> text
'spÄm'
>>> len(text)
4
```

# Files – Unicode encoding text files

To really see what's stored in your file use binary mode

```
S = 'sp\xc4m'
>>> S
'spÄm‘

>>> file = open('unidata.txt', 'w', encoding='utf-8')      # Write/encode UTF-8 text
>>> file.write(S)
4
>>> file.close()

>>> text = open('unidata.txt', encoding='utf-8').read()    # Read/decode UTF-8 text
>>> text
'spÄm'
>>> len(text)
4
```

# Files – Unicode encoding text files

To manually encode and decode

```
>>> text.encode('utf-8')
b'sp\xc3\x84m'
>>> raw.decode('utf-8')
'spÄm'
```

# Files – Further exploration

Spend some time exploring the file functions in python. Open up an interpreter and run the dir call on any open file

```
>>> dir (f)
```

What do you get???

Then run help on some of the method names that come back:

```
>>> help ( f.readlines)
```

# Files – Advanced features of python

For more advanced tasks, though, Python comes with additional file-like tools: pipes, FIFOs, sockets, keyed-access files, persistent object shelves, descriptor-based files, relational and object-oriented database interfaces, and more. Descriptor files, for instance, support file locking and other low-level tools, and sockets provide an interface for networking and interprocess communication.

# Python – Sets

Python includes a data type for **Sets**. Sets, are a recent addition to the language that are neither mappings nor sequences; rather, they are unordered collections of unique and immutable objects.

A set is an unordered collection with no duplicate elements.

Basic uses include membership testing and eliminating duplicate entries.

Set objects also support mathematical operations like **union**, **intersection**, **difference**, and **symmetric difference**.

# Sets – Syntax

To create a set use

{} Curly braces

or the set() function

set()

However, to create an empty set you have to use

set()  # creates an empty set

Do not use

{}  # – because this will create an empty dictionary!

# Sets – Syntax

To create a set use

{} Curly braces

or the set() function

set()

However, to create an empty set you have to use

set()  # creates an empty set

Do not use

{}  # – because this will create an empty dictionary!

# Sets – Example

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'pear', 'orange', 'banana', 'apple'}
>>> 'orange' in basket
True
>>> 'avocado' in basket
False
```

# Sets – Operations

```
a = set('abracadabra')
>>> b = set('alacazam')
>>> a                          # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                      # letters in a but not in b
{'r', 'd', 'b'}
```

# Sets – Operations

```
>>> a | b                          # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                           # letters in both a and b
{'a', 'c'}
>>> a ^ b                          # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

# Sets – Comprehensions

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> c = {x for x in 'abracadabra' if x not in 'abc'}
>>> c
{'d', 'r'}
>>>
```

# User Defined Data Types – Classes

Yet another powerful feature of python is to allow a programmer to define their own types via classes.

Classes define new types of objects that extend the core datatypes available.

# Classes – Example

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # self is the new object
            self.pay  = pay
        def lastName(self):
            return self.name.split()[-1]        # Split string on blanks
        def giveRaise(self, percent):
            self.pay *= (1.0 + percent)         # Update pay in place
```

# Classes – Example

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # self is the new object
            self.pay  = pay
        def lastName(self):
            return self.name.split()[-1]        # Split string on blanks
        def giveRaise(self, percent):
            self.pay *= (1.0 + percent)         # Update pay in place
```

# Classes – Example

Calling the class like a function generates instances of our new type, and the class's methods automatically receive the instance being processed by a given method call

```
>>> bob = Worker('Bob Smith', 50000)
>>> sue = Worker('Sue Jones', 60000)
>>> bob.lastName()
'Smith'
>>> sue.lastName()
'Jones'
>>> sue.giveRaise(.10)
>>> sue.pay
66000.0
```

# Python: Language types

| | Object type | Example literals/creation |
|---|---|---|
| √ | Numbers | 1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction() |
| √ | Strings | 'spam', "Bob's", b'a\x01c', u'sp\xc4m' |
| √ | Lists | [1, [2, 'three'], 4.5], list(range(10)) |
| √ | Dictionaries | {'food': 'spam', 'taste': 'yum'}, dict(hours=10) |
| √ | Tuples | (1, 'spam', 4, 'U'), tuple('spam'), namedtuple |
| √ | Files | open('eggs.txt'), open(r'C:\ham.bin', 'wb') |
| √ | Sets | set('abc'), {'a', 'b', 'c'} |
| √ | Other core types | Booleans, types, None |
| | Program unit types | Functions, modules, classes |

# Python – Key Words

| | | | | |
|---|---|---|---|---|
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass | |
| break | except | in | raise | |

Table 2.1: Python Keywords

# Python Functions

In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name.

In Python  functions behave very differently in Python than they do in compiled languages like C. What follows is a brief introduction to the main concepts behind Python functions

# Python Functions

In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name.

In Python  functions behave very differently in Python than they do in compiled languages like C. What follows is a brief introduction to the main concepts behind Python functions

# Functions concepts

def is executable code. Python functions are written with a new statement, the def. Unlike functions in compiled languages such as C, def is an executable statement—your function does not exist until Python reaches and runs the def. In fact, it's legal (and even occasionally useful) to nest def statements inside if statements, while loops, and even other defs.

# Functions concepts

def creates an object and assigns it to a name.

When Python reaches and runs a def statement, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magic about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined attributes attached to them to record data.

# Functions concepts

**lambda** creates an object but returns it as a result. Functions may also be created with the lambda expression, a feature that allows us to in-line function definitions in places where a def statement won’t work syntactically.

# Functions concepts

return sends a result object back to the caller. When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a return statement; the returned value becomes the result of the function call. A return without a value simply returns to the caller (and sends back None, the default result).

# Functions concepts

**yield** sends a result object back to the caller, but remembers where it left off. Functions known as generators may also use the yield statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time.

# Functions concepts

**global** declares module-level variables that are to be assigned. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a global statement. More generally, names are always looked up in scopes—places where variables are stored—and assignments bind names to scopes.

# Functions concepts

**nonlocal declares enclosing function variables that are to be assigned**. Similarly, the nonlocal statement added in Python 3.X allows a function to assign a name that exists in the scope of a syntactically enclosing def statement. This allows enclosing functions to serve as a place to retain state—information remembered between function calls—without using shared global names.

# Functions concepts

**Arguments are passed by assignment (object reference)**. In Python, arguments are passed to functions by assignment (which, as we've learned, means by object reference). As you'll see, in Python's model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

# Functions concepts

**Arguments are passed by assignment (object reference)**. In Python, arguments are passed to functions by assignment (which, as we've learned, means by object reference). As you'll see, in Python's model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

# Functions concepts

**Arguments are passed by position, unless you say otherwise.** Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function calls can also pass arguments by name with name=value keyword syntax, and unpack arbitrarily many arguments to send with *pargs and **kargs starred-argument notation. Function definitions use the same two forms to specify argument defaults, and collect arbitrarily many arguments received.

# Functions concepts

**Arguments are passed by position, unless you say otherwise.** Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function calls can also pass arguments by name with name=value keyword syntax, and unpack arbitrarily many arguments to send with *pargs and **kargs starred-argument notation. Function definitions use the same two forms to specify argument defaults, and collect arbitrarily many arguments received.

# Functions concepts

Arguments, return values, and variables are not declared. As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As one consequence, a single function can often be applied to a variety of object types—any objects that sport a compatible interface (methods and expressions) will do, regardless of their specific types.

# Functions Example

The def statement creates a function object and assigns it to a name. Its general format is as follows:

```
def name(arg1, arg2,... argN):
    statements
    return value
```

# Functions Assignment

Use the def statement to define a function that converts Celsius to Fahrenheit

hint  F = 9/5 * C + 32

# CS 457

Python Language Finish

Week 10 Day 2

3/26/2015

# Agenda

- Administration
  - None
- Python Language
  - Function Arguments

# Administration

# Python: Language types

| | Object type | Example literals/creation |
|---|---|---|
| √ | Numbers | 1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction() |
| √ | Strings | 'spam', "Bob's", b'a\x01c', u'sp\xc4m' |
| √ | Lists | [1, [2, 'three'], 4.5], list(range(10)) |
| √ | Dictionaries | {'food': 'spam', 'taste': 'yum'}, dict(hours=10) |
| √ | Tuples | (1, 'spam', 4, 'U'), tuple('spam'), namedtuple |
| √ | Files | open('eggs.txt'), open(r'C:\ham.bin', 'wb') |
| √ | Sets | set('abc'), {'a', 'b', 'c'} |
| √ | Other core types | Booleans, types, None |
| | Program unit types | Functions, modules, classes |

# Python Functions

In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name.

In Python  functions behave very differently in Python than they do in compiled languages like C. What follows is a brief introduction to the main concepts behind Python functions

# Functions concepts

- **def** creates an object and assigns it to a name and is executable code.
- **lambda** creates an object but returns it as a result
- **return** sends a result object back to the caller
- **yield** sends a result object back to the caller, but remembers where it left off
- **global** declares module-level variables that are to be assigned
- **nonlocal** (3.x) declares enclosing function variables that are to be assigned- Saves state
- **Arguments** are passed by assignment (object reference)
- **Arguments** are passed by position, unless you say otherwise
- **Arguments** are passed by position, unless you say otherwise –operate on any type

# Functions Example

The def statement creates a function object and assigns it to a name. Its general format is as follows:

```
def name(arg1, arg2,... argN):
    statements
    return value
```

# Functions Assignment

Use the def statement to define a function that converts Celsius to Fahrenheit

hint  F = 9/5 * C + 32

# Functions Assignment Solution

Use the def statement to define a function that converts Celsius to Fahrenheit

```
def C2F():
    Celsius = int(input("Enter a temperature in Celsius: "))
    Fahrenheit = 9.0/5.0 * Celsius + 32
    print ("Temperature: " + str(Celsius) + " Celsius = " + str(Fahrenheit), " F")
C2F()
print ("Done")
```

# Functions Assignment Solution

Use the def statement to define a function that converts Celsius to Fahrenheit

```
def C2F():
    Celsius = int(input("Enter a temperature in Celsius: "))
    Fahrenheit = 9.0/5.0 * Celsius + 32
    print ("Temperature: " + str(Celsius) + " Celsius = " + str(Fahrenheit), " F")
C2F()
print ("Done")
```

# Def is a Statement

Def is a statement that executes at runtime.

This is useful for code that needs to be evaluated after it is written, such as localization types of situations.

What if you wanted to write a function that returned the local temperature?

```
def defExample( system, tempIn):
    if system=="Metric":
        def getTemp(tempIn):
            return 9.0/5.0 * tempIn + 32
    else:
        def getTemp(tempIn):
            return 5.0/9.0 * (tempIn - 32)
    return (getTemp(tempIn))

localTemp = defExample("Metric", 100)
print("Local Temp = " + str(localTemp))
```

# Scope in Python

To effectively write functions in Python, one needs to be aware of how python handles variable scope and namespaces.

The scope of a variable (where it can be used) is always determined by where it is assigned in your source code and has nothing to do with which functions call which. This is known as ***Lexical Scoping*** variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a def, it is *local* to that function.
- If a variable is assigned in an enclosing def, it is *nonlocal* to nested functions.
- If a variable is assigned outside all defs, it is *global* to the entire file.

# Scope Details

- **The enclosing module is a global scope**. Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. Global variables become attributes of a module object to the outside world after imports but can also be used as simple variables within the module file itself.
- **The global scope spans a single file only**. Don't be fooled by the word "global" here—names at the top level of a file are global to code within that single file only. There is really no notion of a single, all-encompassing global file-based scope in Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear "global" in Python, think "module."
- **Assigned names are local unless declared global or nonlocal**. By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a global statement inside the function. If you need to assign a name that lives in an enclosing def, as of Python 3.X you can do so by declaring it in a nonlocal statement.

# Scope Details

- **All other names are enclosing function locals, globals, or built-ins**. Names not assigned a value in the function definition are assumed to be *enclosing*scope locals, defined in a physically surrounding def statement; *globals* that live in the enclosing module's namespace; or *built-ins* in the predefined built-ins module Python provides.
- **Each call to a function creates a new local scope**. Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live. You can think of each defstatement (and lambda expression) as defining a new local scope, but the local scope actually corresponds to a function *call*. Because Python allows functions to call themselves to loop—an advanced technique known as *recursion*

# Scope Summary

Within a def statement:

- Name *assignments* create or change local names by default.
- Name *references* search at most four scopes: local, then enclosing functions (if any), then global, then built-in.
- Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes, respectively.

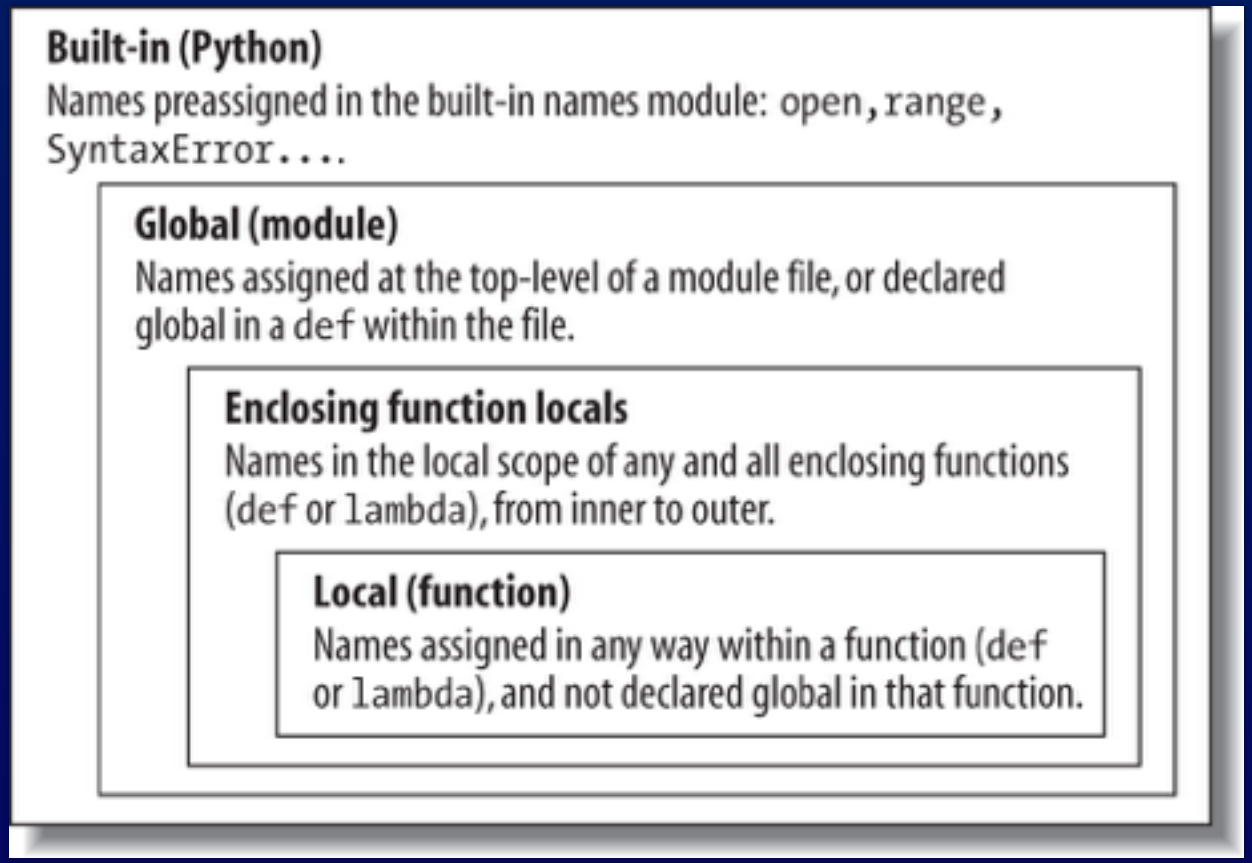
# Scope Lookup Rule

L E (N) G B  rule for simple (non Object) variables.

E in 2.x and N in 3.x

There are three other scopes:

- Temporary loop (Comprehension)
- Exception variables
- Class local variables



# Scope Example

What does this print? Why?

```
X = 99

def func(Y):

    Z = X + Y
    return Z

f = func(1)
print ("f = " + str(f))
```

# Global Statement

Global statements are *namespace declarations*, not type or size declarations.

The global statement tells Python that a function plans to change one or more global names in the enclosing module's scope (namespace).

Note it is good programming practice to ***minimize*** use of Global Statements

# NonLocal Scope

**Nonlocal** declares that a name will be changed in an enclosing scope. Unlike **global**, though, **nonlocal** applies to a name in an enclosing function's scope, not the global module scope outside all defs.

Nonlocal scope names must exist prior to using them.

# NonLocal Example

**What does the following print?**

```
def tester(start):
        state = start              # Referencing nonlocals works normally
        def nested(label):
            nonlocal state
            print(label, state)    # Remembers state in enclosing scope
            state += 1
        return nested
N = tester(0)
print( N("Ham") )

print( N("Spam") )
```

# Function Arguments

We have seen and used arguments to functions earlier. Here ae some key point to keep in mind

- **Arguments are passed by automatically assigning objects to local variable names**. Function arguments —references to (possibly) shared objects sent by the caller—are just another instance of Python assignment at work. Because references are implemented as pointers, all arguments are, in effect, passed by pointer. Objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller**. Argument names in the function header become new, local names when the function runs, in the scope of the function. There is no aliasing between function argument names and variable names in the scope of the caller.
- **Changing a mutable object argument in a function may impact the caller**. On the other hand, as arguments are simply assigned to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Mutable arguments can be input and output for functions.

# Function Arguments

Python is similar to the argument-passing model of the C language (and others) in practice:

- **Immutable arguments are effectively passed "by value."** Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in place anyhow, the effect is much like making a copy.
- **Mutable arguments are effectively passed "by pointer."** Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers—mutable objects can be changed in place in the function, much like C arrays.

# Arguments Matching Basics

Python by default, arguments are matched by *position*, from left to right, and you must pass exactly as many arguments as there are argument names in the function header. However, you can also specify matching by name, provide default values, and use collectors for extra arguments.

But there are some sophisticated tools:

# Arguments Matching Tools

- Positionals
- Keywords:
- Defaults:
- Varargs collecting:
- Varargs unpacking:
- Keyword-only arguments

# Arguments Positionals

- Positionals: matched from left to right

The normal case, which we've mostly been using so far, is to match passed argument values to argument names in a function header by position, from left to right.

func(value1, value 2)

# Arguments Keywords

Keywords: matched by argument name

Alternatively, callers can specify which argument in the function is to receive a value by using the argument's name in the call, with the name=value syntax.

```
def func(spam, eggs, toast=0, ham=0): # First 2 required
    print((spam, eggs, toast, ham))
```

Assignemt What is the output of the following calls?

```
func(1, 2)
func(1, ham=1, eggs=0)
func(spam=1, eggs=0)
func(toast=1, eggs=2, spam=3)
func(1, 2, 3, 4)
```

# Arguments Defaults

Defaults: specify values for optional arguments that aren't passed

Functions themselves can specify default values for arguments to receive if the call passes too few values, again using the name=value syntax.

```
def f(a, b=2, c=3):
    print(a, b, c)


f(1) # Use defaults 1 2 3
>>> f(a=1) 1 2 3
```

# Arguments Varargs collecting

Varargs collecting: collect arbitrarily many positional or keyword arguments

Functions can use special arguments preceded with one or two * characters to collect an arbitrary number of possibly extra arguments. This feature is often referred to as *varargs*, after a variable-length argument list tool in the C language; in Python, the arguments are collected in a normal object.

# Arguments Varargs collecting

The first use, in the function definition, collects unmatched *positional* arguments into a tuple:

```
>>> def f(*args): print(args)
```

When this function is called, Python collects all the positional arguments into a new *tuple* and assigns the variable args to that tuple. Because it is a normal tuple object, it can be indexed, stepped through with afor loop, and so on:

```
>>> f() ()
>>> f(1) (1,)
>>> f(1, 2, 3, 4) (1, 2, 3, 4)
```

# Arguments Varargs collecting

The ** feature is similar, but it only works for *keyword* arguments—it collects them into a new *dictionary*, which can then be processed with normal dictionary tools.

In a sense, the ** form allows you to convert from keywords to dictionaries, which you can then step through with keys calls, dictionary iterators, and the like (this is roughly what the dict call does when passed keywords, but it *returns* the new dictionary):

```
>>> def f(**args): print(args)
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

# Arguments Varargs collecting

Finally, function headers can combine normal arguments, the *, and the ** to implement wildly flexible call signatures.

For instance, in the following, 1 is passed to a by position, 2 and 3 are collected into the pargspositional tuple, and x and y wind up in the kargs keyword dictionary:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Such code is rare, but shows up in functions that need to support multiple call patterns (for backward compatibility, for instance).

# Arguments Varargs unpacking

The * syntax when we call a function, too. In this context, its meaning is the inverse of its meaning in the function definition—it unpacks a collection of arguments, rather than building a collection of arguments. For example, we can pass four arguments to a function in a tuple and let Python unpack them into individual arguments:

```
>>> def func(a, b, c, d): print(a, b, c, d)
>>> args = (1, 2) >>> args += (3, 4)
>>> func(*args)              # Same as func(1, 2, 3, 4)
1 2 3 4
```

# Arguments Varargs unpacking

Similarly, the ** syntax in a function call unpacks a dictionary of key/value pairs into separate keyword arguments:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)            # Same as func(a=1, b=2, c=3, d=4)
1 2 3 4
```

# Arguments Keyword-only arguments

Python 3.X generalizes the ordering rules in function headers to allow us to specify *keyword-only arguments*—arguments that must be passed by keyword only and will never be filled in by a positional argument. This is useful if we want a function to both process any number of arguments and accept possibly optional configuration options.

Note that keyword-only arguments with defaults are optional, but those without defaults effectively become ***required keywords*** for the function

# Arguments Keyword-only arguments

For example, in the following, a may be passed by name or position, b collects any extra positional arguments, and c must be passed by keyword only. In 3.X:

```
def kwonly(a, *b, c):
        print(a, b, c)
>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
```

# Arguments Keyword-only arguments

We can also use a * character by itself in the arguments list to indicate that a function does not accept a variable-length argument list but still expects all arguments following the * to be passed as keywords. In the next function, a may be passed by position or name again, but b and c must be keywords, and no extra positionals are allowed:

```
def kwonly(a, *, b, c):
    print(a, b, c)
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3 >>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
>>> kwonly(1)
TypeError: kwonly() missing 2 required keyword-only arguments: 'b' and 'c'
```

# Arguments Keyword-only arguments

You can still use defaults for keyword-only arguments, even though they appear after the * in the function header. In the following code, a may be passed by name or position, and b and c are optional but must be passed by keyword if used:

```
def kwonly(a, *, b='spam', c='ham'):
    print(a, b, c)
>>> kwonly(1)
1 spam ham
>>> kwonly(1, c=3)
1 spam 3
>>> kwonly(a=1)
1 spam ham
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
```

# Arguments Keyword-only arguments

In fact, keyword-only arguments with defaults are optional, but those without defaults effectively become *required keywords* for the function:

```
>>> def kwonly(a, *, b, c='spam'):
    print(a, b, c)
>>> kwonly(1, b='eggs')
1 eggs spam
>>> kwonly(1, c='eggs')
TypeError: kwonly() missing 1 required keyword-only argument: 'b'
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
```

# Arguments Keyword-only arguments

```
>>> def kwonly(a, *, b=1, c, d=2):
    print(a, b, c, d)
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
```

# Note About Second Assignment

This weekend I will post the second assignment details

We will implement a book cover graphic editing system for Azteca Books.

Stay tuned.