**Python Software Design**

This course is for students with a strong background in programming in C++, such as that provided by two of our courses at USF: Object-Oriented Design and Data Structures. Further software electives would be beneficial. In these notes, we provide an introduction to the basics of Python which tries to leverage the student's previous programming knowledge

**Part I**

**Python Overview**

Python is object-oriented all the way down to its core.

All data, functions, modules, etc. are instances of classes and may have (but are not required to have) attributes and methods. This includes numbers, which in C++ and Jave are not objects and do not have associated methods.

This gives the language a consistency (and useful features) that some other languages lack.

Despite the object-oriented nature of Python, you can write significant programs in a procedural style, without explicitly defining any classes. Of course, you will be using some OOP notation, like the dot notation for accessing attributes of objects you use.

Python has many built-in types and functions, probably more than more traditional languages. And there it has a strong collection of built-in modules defining useful classes.

In addition, there is a wealth of third party modules, providing classes and other code, available online. Modules are made part of program by means of import statements, somewhat similar to C and C++ #include statements.

It is considerably easier to build sophisticated applications in Python than in most other languages.

**Variables**

Variables in languages like C++ and Java have a specified type an are associated with a specific address in memory determined at compile time. Python "variables" do not follow this model. They do not have a declared type; in fact, there is now way to declare the type of a variable.

All Python "variables" are just names for (or references to) objects (values). They are ***bound*** to a value by an assignment statement. The same variable may be rebound to a different object of a different type at any later point in the program.

*Example*: the statement x = 3500 when executed will create an integer object (type int) with value 3500 and make x a reference to that object. And we did not need to make any declaration for x prior to that statement. If we later execute the statement x = "Hi" a string object will create with value "Hi" and x will be bound to that object. If we had not defined any other references to the integer object, it will be made available for garbage collection.

Python *objects* are strongly typed, which means that the type of an object does not change, even if its value is mutable. Python variables are not strongly typed; in fact, variables never have a declared type.

***There are no type declarations in Python***.

For example, when you define a function you do not declare a return type and you do not specify types for the parameters.

When the Python interpreter processes a function definition, it creates a *function object.*  Remember: everything in Python is an object.

Since functions are objects, they may have data attributes. One particular attribute is a "docstring", which is a string written immediately after the first line of the function definition. If F is a function that been given a documentation string in this way, you can retrieve the string as F.\_\_doc\_\_.

This is one of the many "introspective" features of Python. Another is provided by the built-in type function that returns the type of the argument object. For example, type(3.2) will return <class 'float'> and type("hello") will return <class 'str'>.

User defined names: start with a letter or underscore, followed by any number of letters, digits or underscores. It may not be any of the Python reserved words (see page 65 of the pocket reference). As in C++, case matters. User-defined names are created by assignment but must exist when referenced.

It is traditional for user-defined variables to be all lower case with words separated by underscores.

**Number Types and Numerical Operators**

The types int and float are what you would expect. With a few exceptions (see below).

Integers are arbitrary precision. There is no predetermined maximum value. However, when you get outside the machine representable range, operations become slower. The numerical operators are the same as in C and C++ with some exceptions. The / operator is the "true division" operator always returns a float. The notation // is used for integer division and may be used with both integer and float values.

Examples   
9/4 returns 2.25  
9//4 returns 2  
9.0/4 returns 2.25  
9.0//4 returns 2.0

There is an exponential operator: \*\*: 2\*\*3 return 8. The % operator applies to both integers and floats; applied to a float, it returns a float: 10.0 % 3 returns 1.0.  
  
There is a special object **None** that means, essentially, *nothing*. It is the only object of the **NoneType** class . One use of **None** is for functions – it is the default return value of functions. If your code does not have a return statement, **None** is automatically returned.

**Boolean Type (bool)**

Type **bool** has two values: **False** and **True**. When used as integers, False is 0 and True is 1.

Comparison operators use the same notation as in C++: ==, != <, >, <=, >=.

Logical operators

C++ Python

! **not**|| **or**&& **and**

Relational Operators

Python has the usual relational operators – ==, !=, <, <=, >, >= with their normal meanings.

There is one more Python-only relational operator: **is,** which is related to ==.

**x == y** returns **True** if the objects referred to by x and y have the same value.

**x is y** returns **True** if x and y refer to the same object

One caution: comparing a variable to **None** should always use **is**, not **==**.

Another interesting feature of python comparisons: they can be chained. The expression   
(a < b < c) is equivalent to (a < b **and** b < c). To understand any chained comparison, break it apart using the and operator as above.

**Strings (a first look)**

In Python, a string is a sequence of Unicode characters enclosed in quotes. Unlike most other languages, a string literal may be enclosed in single quotes ('), double quotes("), or triple quotes (''' or """).

Inside a single quoted string you can use double quotes and inside double quotes you can use single quotes without escaping (\' or \"). Triple quotes are used to delimit multiline strings and can include single or double quotes. Here are some examples:

>>> r = "This isn't bad at all"  
>>> s = 'He said "hello" and left'  
>>> t = '''He said "hello" and left.  
>>> This isn't bad at all'''  
>>> u = """This is a   
 multiline string"""  
>>> v = "This is a\n multiline string"

After the above, u == v returns True.

Another thing about Python in contrast to C++: **there is no separate character class**. A character is just a length one string.

Python provides two ways to write *format strings* (as in C and C++). That is, strings that include placeholders for non-string objects like integers, floats, etc. The oldest method is very similar to C's format strings and uses % to start a placeholder. Thus you will have placeholders like %d for integers, %f for floats, etc. With these placeholders you may control an object's representation more precisely. The notation is very much like that of C.

All the primitive (or basic) types have a string representation. The built-in function str(x) will return the default string representation of the object referred to by x. Thus, in a format string you may use the %s placeholder for all variables and still get the formatted string.

To use the format string to specify the intended object strings, you follow the string by   
%(x,y, …).

*Example*

>>> i = 5  
>>> price = 2.35  
>>> "The cost of %s items will be %s" %(i,i\*price)  
'The cost of 5 items will be 11.75'

The more modern approach use the *str.format* method, which we will cover in detail later.

**Elementary I/O: the print() and input() function**

print("hello") will print the string hello followed by a newline. print has an parameter (end) allowing you to modify the last thing printed. The output of the following statements

print(1, end = ", ")  
print(2, end = ", ")  
print(3)

will be 1, 2, 3

The input function will print the message you include as an argument, and then input the users response. It always returns a string. However, there are conversion functions that will convert a string to an object corresponding to the string, like int(), float(), …

Example

response = input("Enter an integer: ")  
my\_int = int(response)

But the following would be simpler and more compact:  
my\_int = int(input("Enter an integer: "))

If the user inputs an invalid string, an exception will be raised. ( Python uses raise instead of throw). So this statement should be inside a try-except More on exceptions later.

**Flow of control**

As in C++, a condition is any expression that may be evaluated to True or False. Except that, in our case, evaluated to an object to be *interpreted* in a Boolean context to True or False. As in C++, conditions are used in if and while statements.

*if statements*

One difference with C++ is that Python uses elif instead of else if.

An if statement consists of the following:

• The if keyword

• A condition (that is, an expression that evaluates to True or False)

• A colon

• Starting on the next line, an indented block of code (called the if clause)

This may be followed by zero or more elif clauses (similar format); and an optional else clause (like the if part, but no condition)

Here is a silly example:

if x > 3:  
 print(x)  
 print("is bigger than 3")  
elif x >2:  
 print(x)  
 print("is bigger than 2 but less equal 3")  
else:  
 print(x)  
 print("is less equal to 2)

Note that

* there are no parentheses surrounding the condition
* the if line is terminated by a colon and end of line
* the blocks are not enclosed in curly braces – the block is determined by its indentation level.

Indentation is used throughout Python to define code blocks (called "*suites*" in the Python literature)

*while statement*

The basic Python while statement is essentially the same as in C++ and provides both continue and break statements. The differences are as in the if statements. The first line is of the form while <condition>: and is followed by the indented body of the while.

sum = 0

i = 1

while i < 10:  
 sum += i

i += 1

print(sum)

*The for statement and the range generator*

An *iterable* is a Python collection-like object that presents a sequence-like interface. That is, there is a first element and each element except the last has a next element. Looking ahead, Python lists, tuples and dictionaries are iterables.

A Python for statement consists of the line for k in <iterable>: followed by the indented body of the for loop, where <iterable> is some iterable object. To emulate the typical C++ for(i = a; i < n; ++i), you use the built-in range iterable. For integers i and j, range(i,j) generates, in order, the integers m satisfying i ≤ m < j. We can include a "jump" amount: range(i,j,k) generates the integers as above but separated by k (i, i+k, …)

for n in range(1,6):  
 print(n,end=' ')  
 print() # prints a newline

Output: 1 2 3 4 5

for n in range(1,10,2):  
 print(n,end=' ')  
 print() # prints a newline

Output: 1 3 5 7 9

**Functions (first look)**

A function definition starts with a line of the following form

**def** <function\_name>(<parameter\_list>):

followed by the indented body of the function. The parameter list may be empty, but the parentheses must be present. The first part of the parameter list must be the positional parameters, if any. Parameters with default values are given by keyword parameters. This should be familiar from C++.

def greeting():

print("Hello")

def int\_sum\_upto(n):  
"returns the sum of the integers from 1 to n" ***# docstring for the fc***  
 sum = 0

i = 1

while i <= n:  
 sum += i

i += 1

return sum

def segment\_sum(n,start=1):  
"returns the sum of the integers from 1 to n" ***# docstring for the fc***n

sum = 0

i = start

while i <= start+n:  
 sum += i

i += 1

return sum<

The calls int\_sum\_upto(10) and segment\_sum(10) will both return 45. The call segment\_sum(10,start=2)will return the value 44.

Later we will explain how to specify a variable number of positional and keyword parameters in a function.

***You are now ready to do the Collatz programming assignment.***

**Part II**

**Sequence Types**

A Python type is a *sequence type* if objects of that type are collections whose elements are accessed using zero-based indexing (as in C++ arrays).

The Python sequence types are

*str*  immutable sequence of Unicode characters

*list* mutable sequence of arbitrary object references of possibly mixed types

*tuple* immutable sequence of arbitrary object references of possibly mixed types

Before we get started with the sequence types, we need to discuss a fundamental property of objects: **mutablility**. An object is immutable if it may not be structurally modified in place.

All the primitive types – int, float, bool – are immutable. If variable x refers to an int object with value 3 and you rebind it to an int object with value 5, you do not change the 3 object to a 5 object. You create a new int object with value 5 and rebind x to that object. The int object with value 3 still exists and its value has not changed.

If a sequence is immutable, you may not append a new element to the sequence nor replace or remove any existing element of the sequence.

The immutability of tuples is a bit subtle. While you may not remove or replace any of the references in a tuple, you may modify the target of one of the references if it is mutable. Thus if, say, t[3] is a reference to a list, it is legal to append a new value to t[3]. Why? Because doing so would not change the identity of the list and thus the reference to the list does not change. For example, if t = (1,[2,3],5) we may execute t[1].append(4) without an exception. The tuple still contains the same references, even though one of the referenced objects has changed value (but not identity).

Sequence literals

A *list literal* is a comma separated list of object references enclosed in square brackets, like [1,2,"hello"]. The assignment statement L = [1,2,"hello"] constructs a list object containing two int references and one string reference, then makes L a reference to that object. A *tuple literal* is like a list literal except that the elements are enclosed in parentheses instead of square brackets: t = (1,2,"hello"). One peculiarity: a tuple literal with only one element must include a comma after that element. Example: t = (1,).

String literals (used in the creation of instances of str) are sequences of Unicode characters enclosed in quotes. Unlike most other languages, a string literal may be enclosed in single quotes ('), double quotes("), or triple quotes (''' or """). Inside a single quoted string you can use double quotes and inside double quotes you can use single quotes. Triple quotes are used to delimit multiline strings and can include single or double quotes. Here are some examples:

r = "This isn't bad at all"  
s = 'He said "hello" and left'  
t = '''He said "hello" and left.  
This isn't bad at all'''  
u = """This is a   
 multiline string"""  
v = "This is a\n multiline string"

After the above, u == v returns True.

Another thing about Python in contrast to C++: **there is no separate character class**. A character is just a length one string.

Immutability examples

>>> s = "hello"  
>>> s[4] = '!'  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>

>>> t = (1,"a") # a tuple  
>>> t[0] = "one"  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module  
TypeError: 'tuple' object does not support item assignment

>>> L = [1,"a"] # a list  
>>> L[0] = "one"  
>>> L  
['one', 'a']

The following discussion applies to lists, tuples and strings.

*Indexed Access*

If you attempt to access a sequence x with an out-of-bounds index an exception will be raised. An index is out of bounds if it is negative or is greater than or equal to the sequence length,.

>>> L  
[1, 2, 3, 4, 5, 6]  
>>> len(L)  
6  
>>> L[6]  
Traceback (most recent call last):

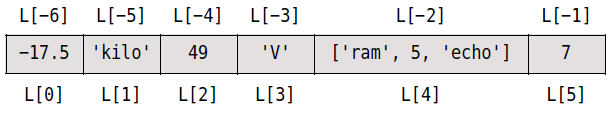
File "<stdin>", line 1, in <module>

IndexError: list index out of range

*Negative Indexing*  
There is a built-in (polymorphic) Python function named len() which returns the length of an object of a sequence object.

One very convenient feature of Python sequence types is the use of negative indices. For example, if x is a sequence, then x[-1] is the last entry in x. More specifically, for integer i with   
1 ≤ i ≤ len(x), x[-i] is the same as x[len(x)-i].

Given the assignment   
L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]  
the list referenced by L is



Slices of a Sequence

Slices are a way of specifying subsequences of a sequence object. Given indexes i and j of sequence x with i ≤ j, x[i:j] is the subsequence of x consisting of all entries from index i up to but not including j. Thus, x[i:i] is always empty. Here are some slices of the list L given above.

>>> L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]  
>>> L[0:len(L)]  
[-17.5, 'kilo', 49, 'V', ['ram', 5, 'echo'], 7]  
>>> L[1:3]  
['kilo', 49]  
>>> L[1:-1]  
['kilo', 49, 'V', ['ram', 5, 'echo']]  
>>> L[-4:-1]  
[49, 'V', ['ram', 5, 'echo']]

The default value for the first slice index is 0. Thus x[:j] is the same as x[0:j]. The default value for the second index is len(x)(recall that the last index is not part of the slice). Thus x[i:] is the sub-list of all entries from index i to the end of the sequence. If both indices are omitted, you get the entire sequence.

For strings and tuples, a slice returns a new object representing the slice. For lists, however, it returns a reference to the sub-list. You can thus modify the list via the slice.

>>> L = [1,2,3,4,5,6]  
>>> M = L  
>>> M[:-1] = []  
>>> M  
[6]

You may also specify slices with a *stride*. The slice x[i:j:k] specifies the elements from index i up to but not including j with a gap of k between the elements:

x[i], x[i+k], x{i+2k], …

>>> L = [1,2,3,4,5,6]

>>> L[1:5:2]

[2, 4]

>>> L[1:6:2]

[2, 4, 6]  
6

You may only replace a "strided" slice by a sequence of the same length. If the lengths differ, an exception will be raised.

>>> L[0:-1:2] = [1] ***# slice consists of every other element of L***  
Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: attempt to assign sequence of size 1 to extended slice of size 3  
  
>>> L[0:-1:2] = [-5,-5,-5]

>>> L

[-5, 2, -5, 4, -5, 6]

For larger slices, it would be better in the above to use the replication operator.

>>> L[0:-1:2] = [-5]\*len(L[0:-1:2])

Special use of slicing: *reversing a list*. First of all, if you use negative striding, the slice   
x[i:j:-1] goes from index  **i-1** down to and including j. The following slice produces the reverse of the list L:

>>> L[len(L):0:-1]  
[6, 5, 4, 3, 2]

If you omit indices with a negative stride, the default first index becomes len(x) and the default second index becomes 0. Thus

>>> L[::-1]  
[6, 5, 4, 3, 2, 1]

*String Concatenation*

Operator overloading in C++ class is done by defining a method whose name consists of the word operator followed by the usual operator symbol (like operator+). Python accomplishes this with "magic methods". To overload the + operator in a class, you define a method named

\_\_add\_\_(). The str class has such a magic method and it returns a new string obtained by concatenating the calling str object and the parameter str object. Note that neither the calling string nor the parameter string are modified – they are immutable, after all.

Example:

>>> s = "Hello"  
>>> t = "There"  
>>> s+t  
'HelloThere'

The \* operator is used to concatenate a string with itself some number of times:

>>> 'Aye'+'Aye'+'Aye'  
'AyeAyeAye'  
  
>>> 'Aye'\*3  
'AyeAyeAye'

**The Dictionary Data Type**

Like a list, a dictionary is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

There are no restrictions on the type of dictionary values. However, because dictionaries are implemented with hashing for quick lookup objects of mutable types like list cannot be used because their hash values will vary as the dictionary is modified. So at the very least, dictionary keys must be immutable. However, there is a little wrinkle with tuples. Essentially, a tuple must be immutable at all depths to be hashable. So (1,[2,3],4) cannot be used as a dictionary key. But this is a weird example and not good practice for tuples.

In code, a dictionary is typed with curly braces, {}. Enter the following into the interactive shell:

>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}

This assigns a dictionary to the myCat variable. This dictionary’s keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

>>> myCat['size']  
'fat'  
>>> 'My cat has ' + myCat['color'] + ' fur.'  
'My cat has gray fur.'

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}

Unlike lists, items in dictionaries are unordered. The first item in a list named spam would be spam[0]. But there is no “first” item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

Example:

>>> spam = ['cats', 'dogs', 'moose']  
>>> bacon = ['dogs', 'moose', 'cats']  
>>> spam == bacon  
False  
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}  
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}  
>>> eggs == ham  
True

Because dictionaries are not ordered, they can’t be sliced like lists. Trying to access a key that does not exist in a dictionary will result in a KeyError exception, much like a list’s “out-of-bounds” IndexError exception. In the following interactive shell command sequence notice the error message that shows up because there is no 'color' key:

>>> spam = {'name': 'Zophie', 'age': 7}  
>>> spam['color']  
Traceback (most recent call last):  
File "<pyshell#1>", line 1, in <module>  
spam['color']  
KeyError: 'color'

Though dictionaries are not ordered, the fact that you can have arbitrary hashable objects for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends’ birthdays. You can use a dictionary with the names as keys and the birthdays as values.

The file *birthdays.py* has been created and filled with this content

birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

while True:

print('Enter a name: (blank to quit)')

name = input()

if name == '':

break

if name in birthdays: ***# is name a current key of the dictionary?***

print(birthdays[name] + ' is the birthday of ' + name)

else:

print('I do not have birthday information for ' + name)

print('What is their birthday?')

bday = input()

birthdays[name] = bday   
 ***# indexing can be used for access or for creating a new entry   
 # in a dictionary***

print('Birthday database updated.')

Example run of the above program with the input shown in italics:

Enter a name: (blank to quit)

*Alice*

Apr 1 is the birthday of Alice

Enter a name: (blank to quit)

*Eve*

I do not have birthday information for Eve

What is their birthday?

*Dec 5*

Birthday database updated.

Enter a name: (blank to quit)

*Eve*

Dec 5 is the birthday of Eve

Enter a name: (blank to quit)

*The keys(), values(), and items() Methods*

There are three dictionary methods that will return list-like values of the dictionary’s keys, values, or both keys and values: keys(), values(), and items(). The values returned by these methods are not true lists: They cannot be modified and do not have an append() method. But these data types (dict\_keys, dict\_values, and dict\_items, respectively) can be used in for loops. To see how these methods work, consider the following interactive shell sequence:

>>> spam = {'color': 'red', 'age': 42}

>>> for v in spam.values():

… print(v)

red

42

Here, a for loop iterates over each of the values in the spam dictionary. A for loop can also iterate over the keys or both keys and values:

>>> for k in spam.keys():

… print(k)

color

age

>>> for i in spam.items():

… print(i)

('color', 'red')  
('age', 42)

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the dict\_items value returned by the items() method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the list() function. Shell example:

>>> spam = {'color': 'red', 'age': 42}

>>> spam.keys()

dict\_keys(['color', 'age'])

>>> list(spam.keys())

['color', 'age']

The list(spam.keys()) line takes the dict\_keys value returned from keys()

and passes it to list(), which then returns a list value of ['color', 'age'].

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Shell example:

>>> spam = {'color': 'red', 'age': 42}

>>> for k, v in spam.items():

… print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42

Key: color Value: red

*Checking Whether a Key or Value Exists in a Dictionary*

Recall from the previous chapter that the in and not in operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Shell example:

>>> spam = {'name': 'Zophie', 'age': 7}

>>> 'name' in spam.keys()

True

>>> 'Zophie' in spam.values()

True

>>> 'color' in spam.keys()

False

>>> 'color' not in spam.keys()

True

>>> 'color' in spam

False

In the previous example, notice that 'color' in spam is essentially a shorter version of writing 'color' in spam.keys(). This is always the case: If you ever want to check whether a value is (or isn’t) a key in the dictionary, you can simply use the in (or not in) keyword with the dictionary value itself.

*The get() Method*

It’s tedious to check whether a key exists in a dictionary before accessing that key’s value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

*Interactive shell example*:

>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'

'I am bringing 2 cups.'

>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'

'I am bringing 0 eggs.'

Because there is no 'eggs' key in the picnic items dictionary, the default value 0 is returned by the get() method. Without using get(), the code would have caused an error message, such as in the next example.

>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

KeyError: 'eggs'

*The setdefault() Method*

You’ll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

spam = {'name': 'Pooka', 'age': 5}

if 'color' not in spam:

… spam['color'] = 'black'

The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key’s value.   
  
>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing %s eggs' %picnicItems['eggs']

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

KeyError: 'eggs'

Interactive shell example:

>>> spam = {'name': 'Pooka', 'age': 5}

>>> spam.setdefault('color', 'black')

'black'

>>> spam

{'color': 'black', 'age': 5, 'name': 'Pooka'}

>>> spam.setdefault('color', 'white')

'black'

>>> spam

{'color': 'black', 'age': 5, 'name': 'Pooka'}

The first time setdefault() is called, the dictionary in spam changes to   
{'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the

value 'black' because this is now the value set for the key 'color'. When

spam.setdefault('color', 'white') is called next, the value for that key is not

changed to 'white' because spam already has a key named 'color'.

The setdefault() method is a nice shortcut to ensure that a key exists.

Here is a short program, in a file named *characterCount.py,* that counts the number of occurrences of each letter in a string.

message = 'It was a bright cold day in April, and the clocks were' message += 'striking thirteen.'

count = {}

for character in message:

count.setdefault(character, 0)

count[character] = count[character] + 1

print(count)

The program loops over each character in the message variable’s string, counting how often each character appears. The setdefault() method call ensures that the key is in the count dictionary (with a default value of 0) so the program doesn’t raise a KeyError exception when the line count[character] = count[character] + 1 is executed.

When you run this program, the output will look like this:

{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}

***You are now ready to do the Inventory Assignment***

**Part III**

**Boolean Context**

Every python object can be used in a "Boolean context", that is, where a logical value is expected. In C++, integers can be used in a Boolean context, where 0 is interpreted as false and any other integer is interpreted as true.

In Python an object of any built-in type may be used in a Boolean context. Integers behave in the same way as in C++. And floats behave the same as integers. For container types like strings, lists, tuples and dictionaries, an empty container is interpreted as False and a nonempty container as True.

For example, given a list object L, instead of writing if L != [], you can be simply write if L.

One of the interesting features of Python is how the value of a Boolean expression is determined. As in C++, short-circuit evaluation is used. As soon as the result of the evaluation is known, further terms are ignored. For all Boolean operators except **not**, the expression returns the object that determined the result (which could then be viewed as True or False). When the **not** operator is applied to a non-boolean object, it always returns either True or False. Here are some examples.

0 **and** 2 returns 0

2 **and** 0 returns 0

0 **or** 2 returns 2

[] **and** 2 returns []

2 **and** [] returns []

(1,2) **or** (2,3) returns (1,2)

(1,2) **and** (2,3) returns (2,3)

-5 **or** 0 returns -5

not 2 returns False

not 0.0 returns True

0 == False returns True

2 == True returns False

1 == True returns True

**Importing Modules**

All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions you’ve seen before. Python also comes with a set of modules called the standard library. Each module is a file containing various related definitions, including object references functions and classes that can be embedded in your programs.

For example, the math module has mathematics- related functions, the random module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an import statement. In code, an import statement consists of the following:

• The import keyword

• The name of the module

• Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the definitions of that module. Let’s give it a try with the random module, which will give us access to the random.randint() function.

Enter this code into the file editor, and save it as *printRandom.py*:

import random

for i in range(5): print(random.randint(1, 10))

When you run this program, the output will look something like this:

4

1

8

4

1

The random.randint() function call evaluates to a random integer value between the two integers that you pass it. Since randint() is in the random module, you must first type random. in front of the function name to tell Python to look for this function inside the random module.

Here’s an example of an import statement that imports four different modules:

import random, sys, os, math

Now we can use any of the functions in these four modules. We’ll learn more about them later in these notes.

The from import Statements

An alternative form of the import statement is composed of the from key- word, followed by the module name, the import keyword, and a list of functions or classes defined in the module. Example:

from random import randint, choices

After this statement, With this form of import statement, calls to the functions randint and choices from random will not need the random. prefix.

You may use all names froom a module M without the random. prefix by an import statement like this:  
  
from M import \*

This is not recommended because you could easily end up with name collisions ("namespace pollution"). The one situation where you will see this is with the *tkinter* GUI module:

from tkinter import \*

If you find a module name too long you can use an import as statement:

import tkinter as tk

Now you can use the prefix tk. in place of tkinter.