Python

Jarek Szlichta
http://data.science.uoit.ca/

Conditions

- **Python uses boolean variables to evaluate conditions**
- The boolean values True and False are returned when an expression is evaluated
  - **variable assignment is done using a single equals operator "=“**
  - **comparison between two variables is done using the double equals operator "==“**
  - The "not equals" operator is marked as "!="

```
x = 2
print x == 2 # prints out True
print x == 3 # prints out False
print x < 3  # prints out True
```

Boolean Operators

- The "and" and "or" boolean operators allow building complex boolean expressions:

```
name = "John"
age = 23
if name == "John" and age == 23:
    print "Your name is John, and you are also
      23 years old."
if name == "John" or name == "Rick":
    print "Your name is either John or Rick."
```

The "in" Operator

- **The "in" operator could be used to check if a specified object exists within an iterable object container**
  - such as a list:

```
if name in ["John", "Rick"]:
    print "Your name is either
      John or Rick."
```

Code Blocks

- **Python uses indentation to define code blocks, instead of brackets**

```
if <statement is true>:
   <do something>
   ....
   ....
elif <another statement is true>:
# else if
   <do something else>
   ....
   ....
else:
   <do another thing>
   ....
   ....
```

Code Blocks Example

- Test if x equals to 2

```
x = 2
If x == 2:
    print "x equals two!"
else:
    print "x does not equal to two."
```

The 'is' and 'not' Operators

- Unlike the"==“operator, the "is" operator does not match the values of the variables, but the instances themselves

```
x = [1,2,3]
y = [1,2,3]
print x == y # Prints out True
print x is y # Prints out False
```

- Using "not" before a boolean expression inverts it

```
print not False # Prints out True
print (not False) == (False) # Prints out False
```

Loops

- There are two types of loops in Python, for and while
- **For loops iterate over a given sequence**

```
primes = [2, 3, 5, 7]
   for prime in primes:
        print prime
```

- **For loops can iterate over a sequence of numbers using the "range" and "xrange" functions**
  - **the range function returns a new list with numbers of that specified range, whereas xrange returns an iterator,**
  - Iterator (xrange) is more efficient

Xrange Example

```
# Prints out the numbers 0,1,2,3,4
for x in xrange(5): # or range(5)
    print x

# Prints out 3,4,5
for x in xrange(3, 6): # or range(3, 6)
    print x

# Prints out 3,5,7
for x in xrange(3, 8, 2):
# or range(3, 8, 2)
  print x
```

"while" Loops

- **While loops repeat as long as a certain boolean condition is met:**

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
    print count
    # This is the same as count = count + 1
    count += 1
```

"break" Statement

- **Break is used to exit a for/while loop**

```
#Prints out 0,1,2,3,4
count = 0
while True:
    print count
    count += 1
    if count >= 5:
        break
```

" continue" Statement

- **Continue is used to skip the current block**

```
# Prints out only odd numbers - 1,3,5,7,9
 for x in xrange(10):
     # Check if x is even
     if x % 2 == 0:
         continue
     print x
```

Can We Use "else" Clause for Loops?

- **unlike languages like C,CPP.. we can use else for loops**

```
# Prints out 0,1,2,3,4
#and then it prints count value reached 5

count=0
while(count<5):
    print count
    count +=1
else:
    print "count value reached %d"
      %(count)
```

Functions

- Functions are a convenient way to divide your code into useful blocks
  - allowing us to order our code
  - make it more readable
  - reuse it and save some time
- Also functions are a key way to define interfaces so programmers can share their code (e.g., GitHub)

How do you write functions in Python?

- **Functions in python are defined using the block keyword "def", followed with the function's name as the block's name.**

```
def my_function():
    print "Hello From My Function!"
```

- **Functions may**
  - **also receive arguments (variables passed from the caller to the function).**
  - **return a value to the caller, using the keyword- 'return‘**

```
def sum_two_numbers(a, b):
    return a + b
```

How do you call functions in Python?

- **Write the function's name followed by (), placing any arguments within the brackets**

```
# print a simple greeting
my_function()

# after this line x will hold the value 3!
x = sum_two_numbers(1,2)
```

Classes and Objects

- Objects are an encapsulation of variables and functions into a single entity
  - Classes are essentially a template to create your objects
  - Objects get their variables and functions from classes

Basic Class

- A basic class looks like this

```
class MyClass:
    variable = "blah"
    def function(self):
        print "This is a message inside
          the class."
```

- **To assign the above class(template) to an object you would do the following:**

```
myobjectx = MyClass()
```

Accessing Object Variables

- To access the variable inside of the newly created object "myobjectx" do the following:

```
myobjectx.variable
```

- You can create multiple different objects that are of the same class

```
myobjecty = MyClass()
myobjecty.variable = "yackity"
```

- Then print out both values:

```
# This would print "blah"
print myobjectx.variable
# This would print "yackity"
print myobjecty.variable
```

Accessing Object Functions

- To access a function inside of an object you use notation similar to accessing a variable:

```
myobjectx.function()
```

- The above would print out the message, "This is a message inside the class."

Dictionaries

- **A dictionary is a data type similar to arrays, but works with keys and values instead of indexes**
  - Each value stored in a dictionary can be accessed using a key, which is any type of object
    - a string,
    - a number,
    - a list, etc.

    instead of using its index to address it.

Storing Dictionaries

- For example, a database of phone numbers could be stored using a dictionary like this:

```
phonebook = {}
phonebook["John"] = 938477566
phonebook["Jack"] = 938377264
phonebook["Jill"] = 947662781
```

- Alternatively, a dictionary can be initialized with the same values in the following notation:

```
phonebook = {
    "John" : 938477566,
    "Jack" : 938377264,
    "Jill" : 947662781
}
```

Iterating over Dictonaries

- **Dictionaries can be iterated over**

```
for name, number in phonebook.iteritems():
print "Phone number of %s is %d"
     % (name, number)
```

- **To remove a specified index, use either one of the following notations**

```
del phonebook["John"]
```

or

```
phonebook.pop("John")
```

Modules

- **Modules in Python are simply Python files with the .py extension, which implement a set of functions.**
- The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once.
  - If another module in your code imports the same module again, it will not be loaded twice but once only
    - so local variables inside the module act as a "singleton" - they are initialized only once

Modules

- **Modules are imported from other modules using the import command.**

```
# import the library
import urllib
# use it
urllib.urlopen(...)
```

- You can check out the full list of built-in modules in the Python standard library
  - https://docs.python.org/2/library/

Writing Modules

- Writing Python modules is very simple.
- To create a module of your own,
  - create a new .py file with the module name,
  - and then import it using the Python file name (without the .py extension) using the import command.

Generators, List Comprehensions, Regular Expressions, Exceptions and Serialization

Generators

- **Generators are used to create iterators**
  - **Generators are functions which return an iterable set of items, one at a time**
- When an iteration over a set of item starts using the for statement, the generator is run
  - **Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the for loop, returning a new value from the set**

Example of Generator Function

- Here is an example of a generator function which returns 6 random integers:

```
import random

def lottery():
    # returns 6 numbers between 1 and 40
    for i in xrange(6):
        yield random.randint(1, 40)

for random_number in lottery():
    print "And the next number is... %d!"
        % random_number
```

Sample Output

Output Window

```
And the next number is... 21!
And the next number is... 17!
And the next number is... 2!
And the next number is... 40!
And the next number is... 7!
And the next number is... 20!
```

Powered by Sphere Engine ™

List Comprehensions

- **List Comprehensions is a very powerful tool**
  - **It creates a new list based on another list, in a single, readable line**
- Let's say we need to create a list of integers which specify the length of each word in a certain sentence,
  - but only if the word is not the word "the".

Without List Comprehensions

```
sentence = "the quick brown fox jumps over the lazy dog"
   words = sentence.split()
   word_lengths = []
   for word in words:
       if word != "the":
           word_lengths.append(len(word))
```

With List Comprehension

- **With a list comprehension, we could simplify this process to this notation:**

```
sentence = "the quick brown fox jumps
     over the lazy dog"
words = sentence.split()
word_lengths = [len(word) for word in
     words if word != "the"]
```

Fixed Function Arguments

- Every function in Python receives a predefined number of arguments, if declared normally, like this:

```
def myfunction(first, second, third):
    # do something with the 3 variables
    ...
```

Multiple Function Arguments

- It is possible to declare functions which receive a variable number of arguments, using the following syntax:

```
def foo(first, second, third, *therest):
   print "First: %s" % first
   print "Second: %s" % second
   print "Third: %s" %
   third
   print "And all the rest... %s"
         % list(therest)
```

Output

- So calling foo(1,2,3,4,5) will print out:

```
First: 1
Second: 2
Third: 3
And all the rest... [4, 5]
```

Regular Expressions

- **Regular Expressions (sometimes shortened to regexp, regex, or re) are a tool for matching patterns in text.**
  - The applications for regular expressions are wide-spread, but they are fairly complex

Regular Expressions

- An example regex is r"^(From|To|Cc).*?python-list@python.org“
  - the caret ^ matches text at the beginning of a line
  - the part with (From|To|Cc) means that the line has to start with one of the words that are separated by the pipe |.
    - That is called the OR operator, and the regex will match if the line starts with any of the words in the group.
  - The .*? means to un-greedily match any number of characters, except the newline \n character.
    - The un-greedy part means to match as few repetitions as possible (match shortest possible string)
  - The . character means any non-newline character, the * means to repeat 0 or more times, and the ? character makes it un-greedy.

Example

- So, the following lines would be matched by that regex:
  - From: python-list@python.org
  - To: !asp]<,. python-list@python.org
- A complete reference for the re syntax is available at the python docs:
  - https://docs.python.org/3/library/re.html#regular-expression-syntax "RE syntax

Exception Handling

- When programming, errors happen.
  - It's just a fact of life...
    - Perhaps the user gave bad input..
    - Maybe a network resource was unavailable.. Maybe the program ran out of memory..
    - Or the programmer may have even made a mistake!

Exceptions

- Python's solution to errors are exceptions
- You might have seen an exception before:

```
>>> print a
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Try/Exceptions Handling

- **But sometimes you do not want exceptions to completely stop the program.**
  - You might want to do something special when an exception is raised
  - **This is done in a *try/except* block**

Example

- Suppose you're iterating over a list.
  - You need to iterate over 20 numbers, but the list is made from user input, and might not have 20 numbers in it
  - After you reach the end of the list, you just want the rest of the numbers to be interpreted as a 0

Exception Handling Code

```
def do_stuff_with_number(n):
    print n

the_list = (1, 2, 3, 4, 5)

for i in range(20):
   try:
      do_stuff_with_number(the_list[i])
   # Raised when accessing
   # a non-existing index of a list
   except IndexError:
      do_stuff_with_number(0)
```

Sets

- **Sets are lists with no duplicate entries.**

```
print set("my name is Eric and Eric is
    my name".split())
```

- This will print out a list containing "my", "name", "is", "Eric", and finally "and".
  - Since the rest of the sentence uses words which are already in the set, they are not inserted twice.

Operations over Sets

- **Sets allow to calculate intersections and differences**
  - For example, say:

```
a = set(["Jake", "John", "Eric"])
b = set(["John", "Jill"])
```

- **To find out which members attended both events, you may use the "intersection" method:**

```
>>> a.intersection(b)
set(['John'])
```

Difference and Union

- **To find out which members attended only one of the events, use the "symmetric_difference" method:**

```
>>> a.symmetric_difference(b)
set(['Jill', 'Jake', 'Eric'])
```

- **To find out which members attended only one event and not the other, use the "difference" method:**

```
>>> a.difference(b)
set(['Jake', 'Eric']
```

- **To receive a list of all participants, use the "union" method:**

```
>>> a.union(b)
set(['Jill', 'Jake', 'John', 'Eric'])
```

Serialization

- **Serialization is the process of translating data structures or object state into a format that can be stored**
  - for example, in a file or memory buffer, or transmitted across a network connection link
- **Python provides built-in JSON libraries to encode and decode JSON.**
  - **Example JSON object: {"firstName":"John", "lastName":"Doe"}**
- In order to use the json module, it must first be imported:

```
import json
```

- There are two basic formats for JSON data. Either in a string or the object data structure.

Load and Dumps

- **To encode a data structure to JSON, use the "dumps" method.** This method takes an object and returns a String (`import json`):

```
json_string =
json.dumps([1, 2, 3, "a", "b", "c"])
```

- To load JSON back to a data structure, use the "loads" method.
  - This method takes a string and turns it back into the json object data structure:

```
#[1, 2, 3, u'a', u'b', u'c']
print json.loads(json_string)
```

Partial Functions

- You can create partial functions in python by using the partial function from the functools library
  - Partial functions allow to derive a function with x parameters to a function with fewer parameters
  - Import required:

```
from functools import partial
```

Example

- Example: from functools import partial
  - ```
    from functools import partial

     def multiply(x,y): return x * y
     # create a new function
     # that multiplies by 2
     dbl = partial(multiply,2)
     print dbl(4)
    ```
- This code will return 8.
- An important note:
  - the default values will start replacing variables from the left. The 2 will replace x. y will equal 4 when dbl(4) is called.
    ```

Code Introspection

- Code introspection is the ability to examine classes, functions and keywords to know what they are, what they do and what they know.
  - Python provides several functions and utilities for code introspection
    - help()
    - dir() id()
    - type()
    - issubclass()
    - isinstance()

Reading List

- **Review Slides**
- **Recommended**
  - Python Tutorial
    - http://www.learnpython.org/
- **Optional**
  - History and General Information
    - https://en.wikipedia.org/wiki/History_of_Python
    - https://www.python.org/doc/essays/comparisons
    - http://www.programmerinterview.com/index.php/general-miscellaneous/whats-the-difference-between-a-compiled-and-an-interpreted-language/
  - Documentation
    - https://docs.python.org/2/tutorial/

Appendix

- Perform Exercise from the attached appendix
  - This is important if this is your first contact with Python!
  - You can find and do all the attached exercises at:
    - http://www.learnpython.org/

Exercise

- Change the variables in the first section, so that each if statement resolves as True.

Code Window

```
1  # change this code
2  number = 10
3  second_number = 10
4  first_array = []
5  second_array = [1,2,3]
6
7  if number > 15:
8      print "1"
9
10 if first_array:
11     print "2"
12
13 if len(second_array) == 2:
14     print "3"
15
16 if len(first_array) + len(second_array) == 5:
17     print "4"
18
19 if first_array and first_array[0] == 1:
20     print "5"
21
22 if not second_number:
23     print "6"
```

Output Window | Expected Output

```
1
2
3
4
5
6
```

Solution

Code Window

Solution

```
1 # change this code
2 number = 16
3 second_number = 0
4 first_array = [1,2,3]
5 second_array = [1,2]
6
7 if number > 15:
8     print "1"
9
10 if first_array:
11     print "2"
12
13 if len(second_array) == 2:
14     print "3"
```

Exercise

- Loop through and print out all even numbers from the numbers list in the same order they are received.
  - Don't print any numbers that come after 237 in the sequence

Code Window

```
1 numbers = [
2     951, 402, 984, 651, 360, 69, 408, 319, 601, 485, 980, 507, 725, 54
3     615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984, 592, 23
4     386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 61
5     399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 9
6     815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831,
7     958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126, 72
8     743, 527
9 ]
10
11 # your code goes here
```

Output Window | Expected Output

```
402
984
360
408
980
544
390
984
592
236
942
386
462
418
344
236
566
978
328
162
758
918
```

Solution

Code Window

Solution

```
 7      958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126,
 8      743, 527
 9 ]
10
11 # your code goes here
12 for number in numbers:
13     if number == 237:
14         break
15
16     if number % 2 == 1:
17         continue
18
19     print number
```

Exercise

- Add a function named list_benefits() that returns the following list of strings: "More organized code", "More readable code", "Easier code reuse", "Allowing programmers to share and connect code together “
- Add a function named build_sentence(info) which receives a single argument containing a string and returns a sentence starting with the given string and ending with the string " is a benefit of functions!"

Exercise

Code Window

```
1  # Modify this function to return a list of strings as defined above
2  def list_benefits():
3      pass
4
5  # Modify this function to concatenate to each benefit - " is a benef
6  def build_sentence(benefit):
7      pass
8
9  def name_the_benefits_of_functions():
10     list_of_benefits = list_benefits()
11     for benefit in list_of_benefits:
12         print build_sentence(benefit)
13
14 name_the_benefits_of_functions()
15
16
```

Output Window

Expected Output

```
More organized code is a benefit of functions!
More readable code is a benefit of functions!
Easier code reuse is a benefit of functions!
Allowing programmers to share and connect code together is a benefit of fu
```

Solution

Code Window

Solution

```
1 # Modify this function to return a list of strings as defined above
2 def list_benefits():
3     return "More organized code", "More readable code", "Easier code
4
5 # Modify this function to concatenate to each benefit - " is a benef
6 def build_sentence(benefit):
7     return "%s is a benefit of functions!" % benefit
8
9
10 def name_the_benefits_of_functions():
11     list_of_benefits = list_benefits()
12     for benefit in list_of_benefits:
13         print build_sentence(benefit)
14
15 name_the_benefits_of_functions()
```

Exercise

- We have a class defined for vehicles.
- Create two new vehicles called car1 and car2.
- Set car1 to be a red convertible worth $60,000 with a name of Fer, and car2 to be a blue van named Jump worth $10,000

Exercise

Code Window

```
1 class Vehicle:
2     name = ""
3     kind = "car"
4     color = ""
5     value = 100.00
6     def description(self):
7         desc_str = "%s is a %s %s worth $%.2f."
8         % (self.name, self.color, self.kind, self.value)
9         return desc_str
10 # your code goes here
11
12 # test code
13 print car1.description()
14 print car2.description()
```

Output Window

Expected Output

```
Fer is a red convertible worth $60000.00.
Jump is a blue van worth $10000.00.
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
14 car1.color = "red"
15 car1.kind = "convertible"
16 car1.value = 60000.00
17
18 car2 = Vehicle()
19 car2.name = "Jump"
20 car2.color = "blue"
21 car2.kind = "van"
22 car2.value = 10000.00
23
24 # test code
25 print car1.description()
26 print car2.description()
```

Exercise

- Add "Jake" to the phonebook with the phone number 938273443, and remove Jill

Code Window

```
1 phonebook = {
2     "John" : 938477566,
3     "Jack" : 938377264,
4     "Jill" : 947662781
5 }
6
7 # write your code here
8
9
10 # testing code
11 if "Jake" in phonebook:
12     print "Jake is listed in the phonebook."
13 if "Jill" not in phonebook:
14     print "Jill is not listed in the phonebook."
```

Output Window

Expected Output

```
Jake is listed in the phonebook.
Jill is not listed in the phonebook.
```

Powered by Sphere Engine ™

Solution

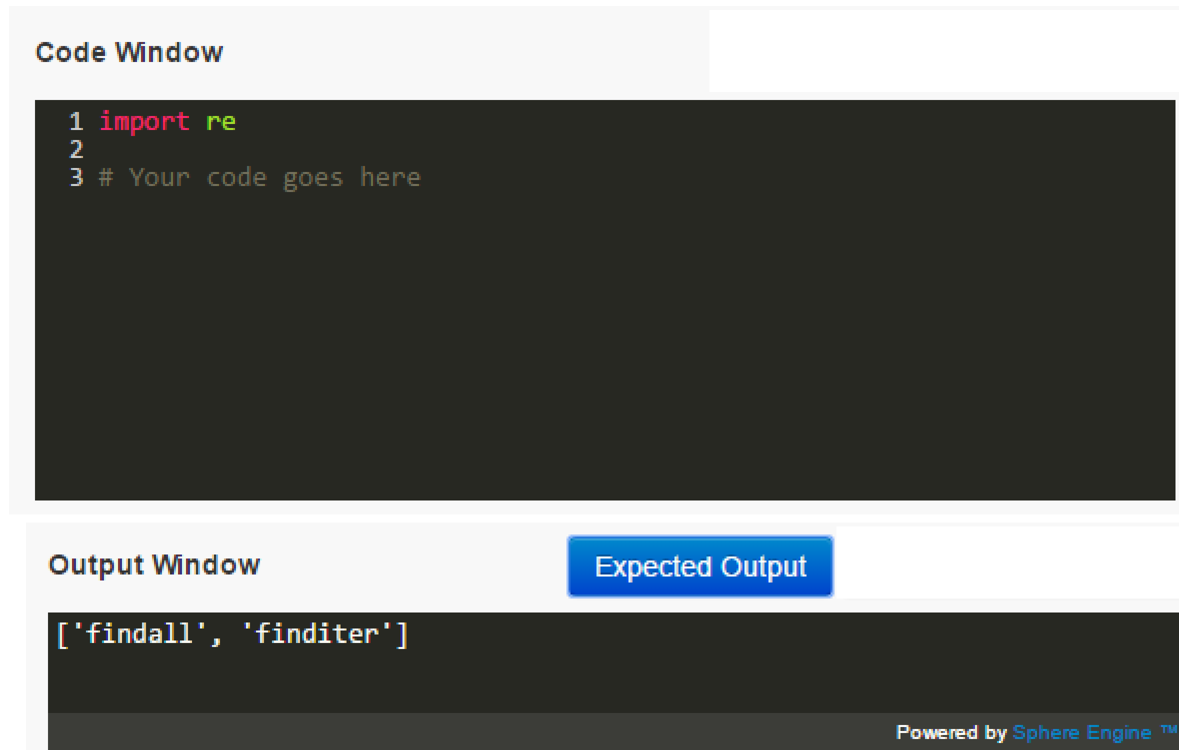
Code Window

Solution

```
1 phonebook = {
2     "John" : 938477566,
3     "Jack" : 938377264,
4     "Jill" : 947662781
5 }
6 # write your code here
7 phonebook["Jake"] = 938273443
8 del phonebook["Jill"]
9
10 # testing code
11 if "Jake" in phonebook:
12     print "Jake is listed in the phonebook."
13 if "Jill" not in phonebook:
14     print "Jill is not listed in the phonebook."
```

Exercise

- In this exercise, you will need to print an alphabetically sorted list of all functions in the re module, which contain the word find.



Solution

Code Window

Solution

```
1 import re
2
3 # Your code goes here
4 find_members = []
5 for member in dir(re):
6     if "find" in member:
7         find_members.append(member)
8
9 print sorted(find_members)
```

Exercise

- Write a generator function which returns the Fibonacci series
  - First 10 elements
- The first two numbers of the series is always equal to 1, and each consecutive number returned is the sum of the last two numbers
  - Use only 2 variables in the generator function

Exercise

Code Window

```
1 # fill in this function
2 def fib():
3     pass #this is a null statement which does nothing when executed,
4 
5 # testing code
6 import types
7 if type(fib()) == types.GeneratorType:
8     print "Good, The fib function is a generator."
9 
10    counter = 0
11    for n in fib():
12        print n
13        counter += 1
14        if counter == 10:
15            break
```

Output Window

Expected Output

```
Good, The fib function is a generator.
1
1
2
3
5
8
13
21
34
55
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
1  # fill in this function
2  def fib():
3      a, b = 1, 1
4      while 1:
5          yield a
6          a, b = b, a + b
7
8  # testing code
9  import types
10 if type(fib()) == types.GeneratorType:
11     print "Good, The fib function is a generator."
12
13     counter = 0
14     for n in fib():
```

Exercise

- In the exercise below, use the given lists to print out a set containing all the participants from event A which did not attend event B.

Code Window

```
1 a = ["Jake", "John", "Eric"]
2 b = ["John", "Jill"]
```

Output Window   Expected Output

```
set(['Jake', 'Eric'])
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
1 a = ["Jake", "John", "Eric"]
2 b = ["John", "Jill"]
3
4 A = set(a)
5 B = set(b)
6
7 print A.difference(B)
```

Exercise

- The aim of this exercise is to print out the JSON string with key-value pair "Me" : 800 added to it.

Code Window

```
3 # fix this function, so it adds the given name
4 # and salary pair to salaries_json, and return it
5 def add_employee(salaries_json, name, salary):
6     # Add your code here
7
8     return salaries_json
9
10 # test code
11 salaries = '{"Alfred" : 300, "Jane" : 400 }'
12 new_salaries = add_employee(salaries, "Me", 800)
13 decoded_salaries = json.loads(new_salaries)
14 print decoded_salaries["Alfred"]
15 print decoded_salaries["Jane"]
16 print decoded_salaries["Me"]
```

Output Window   Expected Output

```
300
400
800
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
 4 # and salary pair to salaries_json, and return it
 5 def add_employee(salaries_json, name, salary):
 6     salaries = json.loads(salaries_json)
 7     salaries[name] = salary
 8 
 9     return json.dumps(salaries)
10 
11 # test code
12 salaries = '{"Alfred" : 300, "Jane" : 400 }'
13 new_salaries = add_employee(salaries, "Me", 800)
14 decoded_salaries = json.loads(new_salaries)
15 print decoded_salaries["Alfred"]
16 print decoded_salaries["Jane"]
17 print decoded_salaries["Me"]
```

Exercise

- Edit the function provided by calling partial() and replacing the first three variables in func().
  - Then print with the new partial function using only one input variable so that the output equals 60.

Code Window

```
1 #Following is the exercise, function provided:
2 from functools import partial
3 def func(u,v,w,x):
4     return u*4 + v*3 + w*2 + x
5 #Enter your code here to create and print with your partial function
```

Output Window | Expected Output

```
60
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
1 from functools import partial
2 def func(u,v,w,x):
3     return u*4 + v*3 + w*2 + x
4 
5 p = partial(func,5,6,7)
6 print p(8)
```

Exercise

- Fill in the foo function so they can receive a variable amount of arguments (3 or more)
  - The foo function must return the amount of extra arguments received.

Code Window

```
1 # edit the functions prototype and implementation
2 def foo(a, b, c):
3     pass
4
5 # test code
6 if foo(1,2,3,4) == 1:
7     print "Good."
8 if foo(1,2,3,4,5) == 2:
9     print "Better."
10
```

Output Window

Expected Output

```
Good.
Better.
Great.
Awesome!
```

Powered by Sphere Engine ™

Solution

Code Window

```
1 # edit the functions prototype and implementation
2 def foo(a, b, c, *args):
3     return len(args)
4
5 # test code
6 if foo(1,2,3,4) == 1:
7     print "Good."
8 if foo(1,2,3,4,5) == 2:
9     print "Better."
```

Exercise

- Fix all the exceptions

Code Window

```
1 # Handle all the exceptions!
2 #Setup
3 actor = {"name": "John Cleese", "rank": "awesome"}
4
5 #Function to modify, should return the last name of the actor
6     return actor["last_name"]
7
8 #Test code
9 get_last_name()
10 print "All exceptions caught! Good job!"
11 print "The actor's last name is %s" % get_last_name()
```

Output Window

Expected Output

```
All exceptions caught! Good job!
The actor's last name is Cleese
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
1 actor = {"name": "John Cleese", "rank": "awesome"}
2
3 def get_last_name():
4     return actor["name"].split()[1]
5
6 get_last_name()
7 print "All exceptions caught! Good job!"
8 print "The actor's last name is %s" % get_last_name()
```

Exercise

- Using a list comprehension, create a new list called "newlist" out of the list "numbers", which contains only the positive numbers from the list, as integers.

Code Window

```
1 numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]
2 newlist = []
3
4 print newlist
```

Output Window   Expected Output

```
[34, 44, 68, 44, 12]
```

Powered by Sphere Engine ™

Solution

Code Window

Solution

```
1 numbers = [34.6, -203.4, 44.9, 68.3, -12.2, 44.6, 12.7]
2 newlist = [int(x) for x in numbers if x > 0]
3
4 print newlist
```