Python:
Control Flow, Functions, Object Name Bindings

Hina Arora

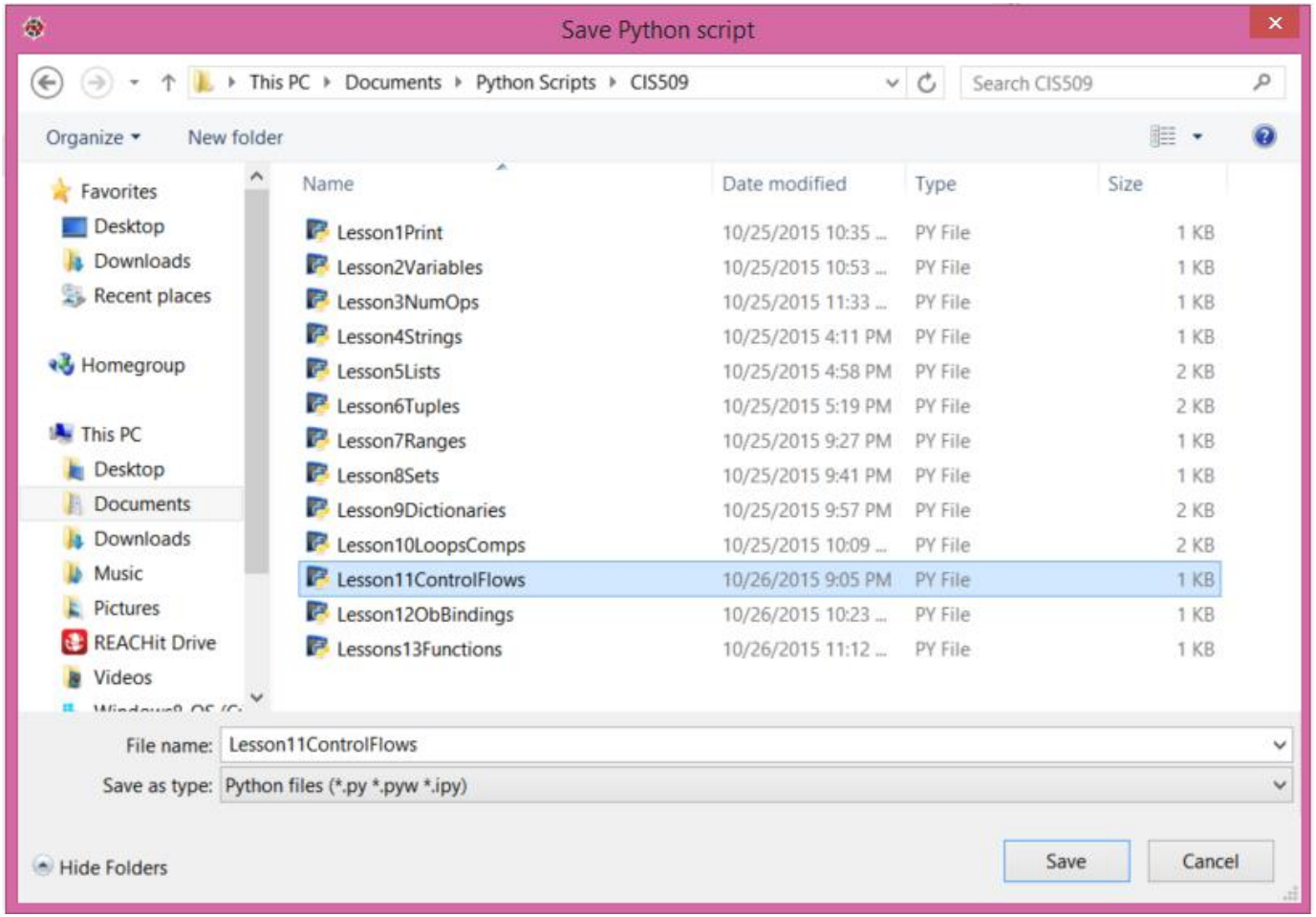
Control Flow

follow along!

(Reference: https://docs.python.org/3/tutorial/controlflow.html)

Create a new file called “Lesson11ControlFlow.py”

- Go to File -> New file…
- Go to File -> Save as…
- Got to CISPy directory
- Save file as “Lesson11ControlFlow.py”



```
print ()

# The while statements will evaluate till the while condition holds true

# let's use a while loop to print out the Fibonacci Series:
#    0, 1, 1, 2, 3, 5, 8, 13, 21, ...
#    so essentially, we start with the elements 0 and 1, and then each
#    successive element of the series is the sum of the previous two elements

print ("Fibonacci series using while loop: ")

a, b = 0, 1

while (b < 100):
    print (b, end = ', ')
    a, b = b, a+b
print ()

print ()

```

```
28 # note: Python uses indentation to identify code blocks
29 # so the following would produce the error:
30 #     "unindent does not match any outer indentation leve"
31
32 print ("Fibonacci series using while loop: ")
33
34 a, b = 0, 1
35
36 while (b < 100):
37   print (b, end = ', ')
38  a, b = b, a+b
39 print ()
40
41 print ()
42
```

```
# The "if" statement first evaluates the "if" condition.
#    - if the condition evaluates to true, it executes the corresponding code
#      block and stops
#    - if the condition evaluates to false, it evaluates the next-in-line
#      "elif" condition, and if that condition evaluates to true, it executes
#      the corresponding code block and stops
#    - if none of the "if" and "elif" conditions are true, it will default to
#      the execution of the code block corresponding to the "else" statement
#      (if one exists) and stop

word = 'Python'
# word = 'Jython'
# word = 'Anaconda'
# word = 'CIS415'
if (word.startswith('P')):
    print ("Word starts with P")
elif (word.endswith('n')):
    print ("Word ends with n")
elif (word.startswith('A')):
    print ("Word starts with A")
else:
    print ("Unknown word")

print ()

```

```
68 # The for statement in Python differs a bit from what you may be used to.
69 # Rather than always iterating over an arithmetic progression of numbers, or
70 # giving you the ability to define both the iteration step and halting
71 # condition, Python's for statement iterates over the items of any iterable
72 # sequence (such as a list, tuple, or string), in the order that they appear
73 # in the sequence
74
75 words = ['cat', 'dog', 'cow', 'parrot', 'hamster', 'goat']
76 for w in words:
77     print (w, len(w))
78
79 print ()
80
```

```
# The range function generates arithmetic prgogressions

# range(n) -> 0,...,n-1 in increments of 1
for i in range(5):
    print (i, end=',')
print ()

```

```
# range(m,n) -> m,...,n-1 in increments of 1
# if m >= n, returns nothing
for i in range(3,10):
    print (i, end=',')
print ()

```

```
# range(m,n,k) -> m,...,<=n-1 in increments of k

# counts in positive increments if n > m, and k is positive
for i in range(3,10,2):
    print (i, end=',')
print ()

# counts in negative increments if n < m, and k is negative
for i in range(10,-30,-5):
    print (i, end=',')
print ()

```

```
106 words = ['jane', 'john', 'mark', 'harry', 'mike', 'ed']
107 wordlist = []
108 for i in range(len(words)):
109     wordlist.append([i, words[i]])
110 print (wordlist)
111 print ()
112
113 print ()
114
```

```
115 # Note: The object returned by range() behaves as if it is a list,
116 # but in fact it isn't. It is an object which returns the successive items of
117 # the desired sequence when you iterate over it, but it doesn't really make
118 # the list, thus saving space.
119
120 print ("range is not a list: ", range(10))
121 print ("generate list underlying range: ", list(range(10)))
122
123 print ()
124
```

```python
# break -
# breaks out of the smallest enclosing for or while loop.
for n in range(1, 10):
    if n % 2 == 0:
        print("found even number: ", n)
        break

print ()

```

```python
# else -
# executed when the loop terminates thro exhaustion of the list (with for),
# or when the condition becomes false (with while),
# but not when the loop is terminated by a break statement

for n in range(1, 10, 2):
    if n % 2 == 0:
        print("found even number: ", n)
        break
else:
    # loop fell through without finding a factor
    print("even number not found")

print()

```

```python
# continue -
# continues with the next iteration of the loop
for n in range(1, 10):
    if n % 2 == 0:
        print("even number: ", n)
        continue
    print("not an even number", n)

print ()

```

```
159 # Pass statements
160 #     the pass statement does nothing. It can be used when a statement is
161 #     required syntactically but the program requires no action.
162
163 #while True:
164 #    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
165
166 #class MyEmptyClass:
167 #    pass  # create a minimal class
168
169 #def initlog(*args):
170 #    pass  # placehodler for function what implementing new code
171
```

What will these statements do?

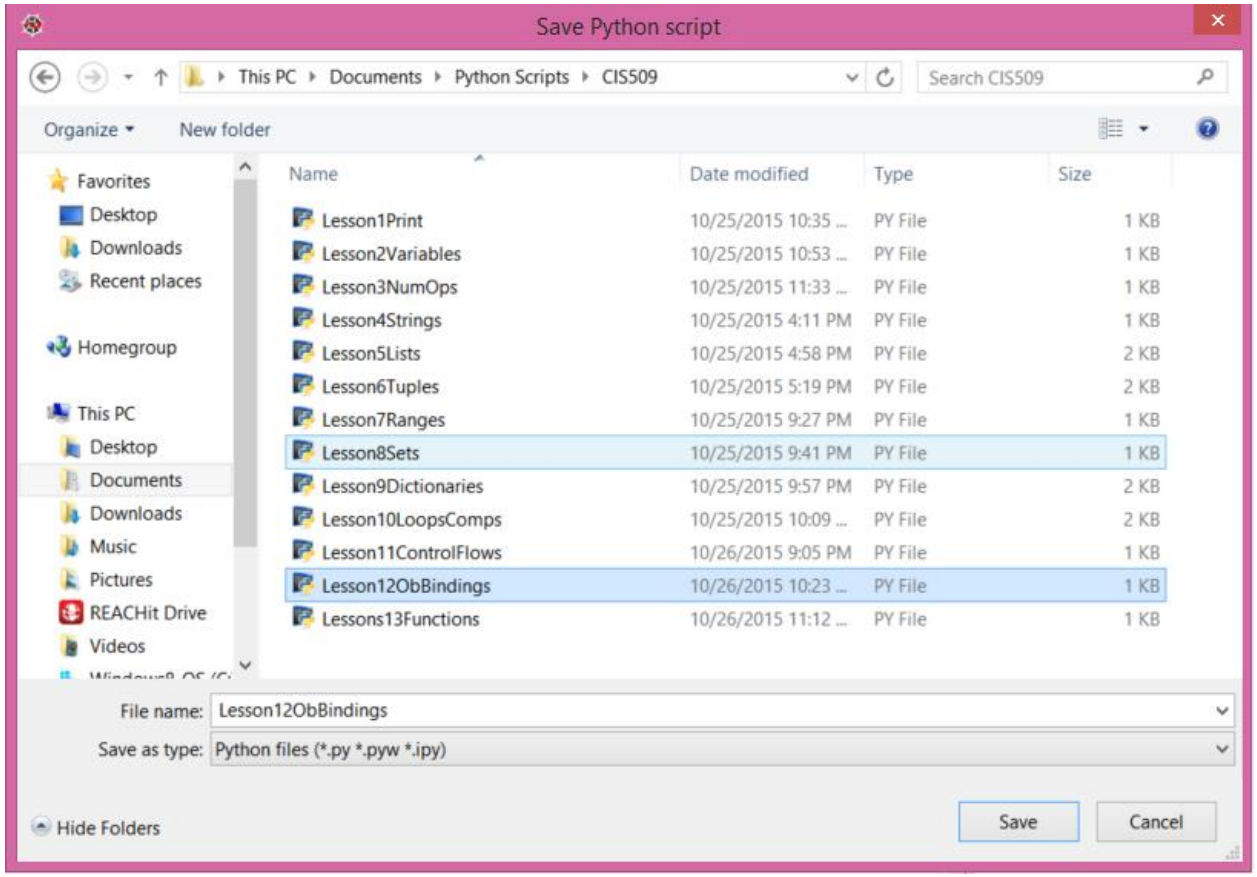
```
172 # test
173
174 for i in range(1, -10, 2):
175     print (i, end = ',')
176
177 while (True):
178     pass
179
180 n = 1
181 while (n < 10):
182  print (n)
183    n = n + 1
184
185 if (5 < 3):
186     print ("condition is true")
```

Detour: Objects, Names and Bindings

(Reference: https://www.jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/)

Create a new file called “Lesson12ObjectBinding.py”

- Go to File -> New file…
- Go to File -> Save as…
- Got to CISPy directory
- Save file as “Lesson12ObjectBinding.py”



- In Python, (almost) everything is an **Object**

- There are two kinds of Objects in Python - Mutable and Immutable

- **Mutable** Object:
  - A Mutable Object exhibits time-varying behavior
  - Changes to a Mutable Object are visible through all Names Bound to it
  - Python's Lists are an example of Mutable Objects

```
# this is allowed
myList = ['Jill', 'Jane', 'Harry']
myList[0] = 'Tom'
```

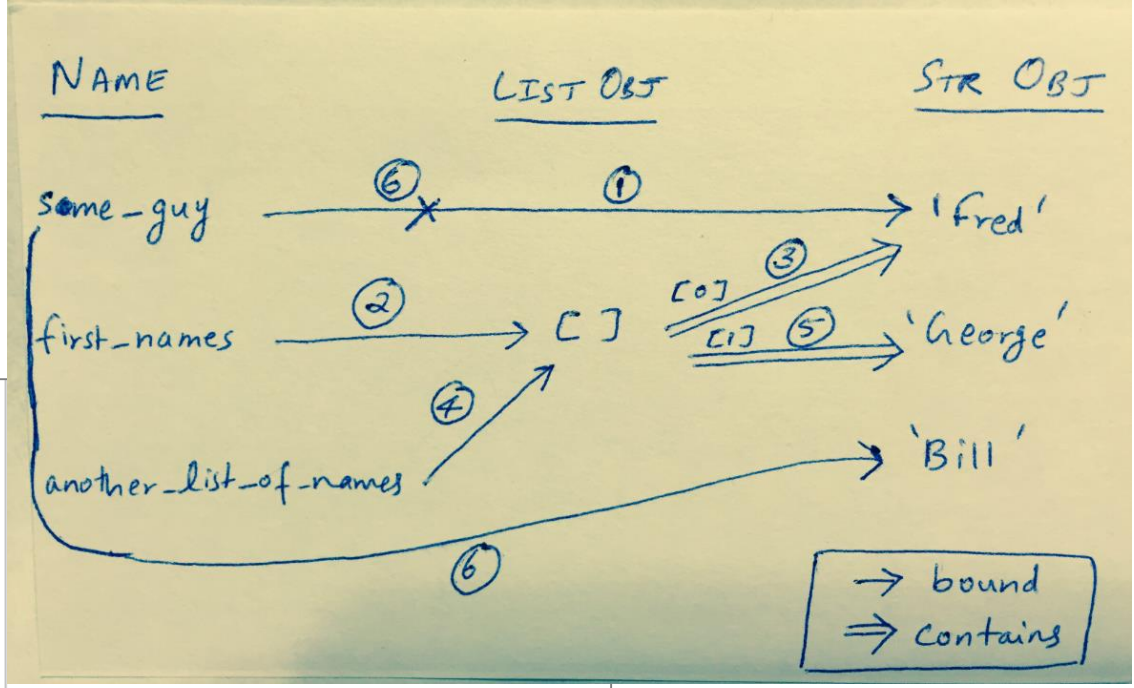
- **Immutable** Object:
  - An Immutable Object does not exhibit time-varying behavior
  - The value of Immutable Objects can not be modified after they are created
  - They can however be used to compute the values of new objects
  - Strings and Integers are examples of Immutable Objects

```
# this is not allowed
myStr = 'Rick'
myStr[0] = 'N'
```

- What we commonly refer to as "variables" in Python are more properly called **Names**

- Likewise, "assignment" is really the **Binding** of a Name to an Object

- Each Binding has a **Scope** that defines its visibility (usually the Block in which the Name originates)




```
# Let's look at an example of Names and Object Binding

# (1) The Name some_guy is Bound to the String Object 'Fred'
some_guy = 'Fred'

# (2) The Name fisrt_names is Bound to the List Object []
first_names = []

# (3) The 0th Element of the List Object created above now contains the String Object 'Fred'
first_names.append(some_guy)

# (4) The Name another_list_of_names is also now Bound to the List Object created above
another_list_of_names = first_names

# (5) The 1st Element of the List Object created above now contains the String Object 'George'
another_list_of_names.append('George')

# (6) The Name some_guy is now Bound to the String Object 'Bill' (and not Bound to String Object 'Fred' anymore)
some_guy = 'Bill'

# So this will now yield 'Bill', ['Fred, 'George'], ['Fred, 'George']
print (some_guy, first_names, another_list_of_names)

# Note: in all, there were 3 Names, 1 List Object, and 3 String Objects in example above

print ()
```

```
# another thing of note is copying versus binding:

# this will simply bind a and b to the same object
# you can test this with b is a (which will evaluate to True below)
# this also means that changes to b will be reflected in a
a = [1, 2, 3]
b = a
print (b is a)
b[2] = 10
print (a, b)

# this will create a copy of a, so a and c now point to different objects
# you can test this with c is a (which will evaluate to False below)
# this also means that changes to c will not be reflected in a
a = [1, 2, 3]
c = []
c[:] = a [:]
print (c is a)
c[2] = 10
print (a, c)

print ()
```

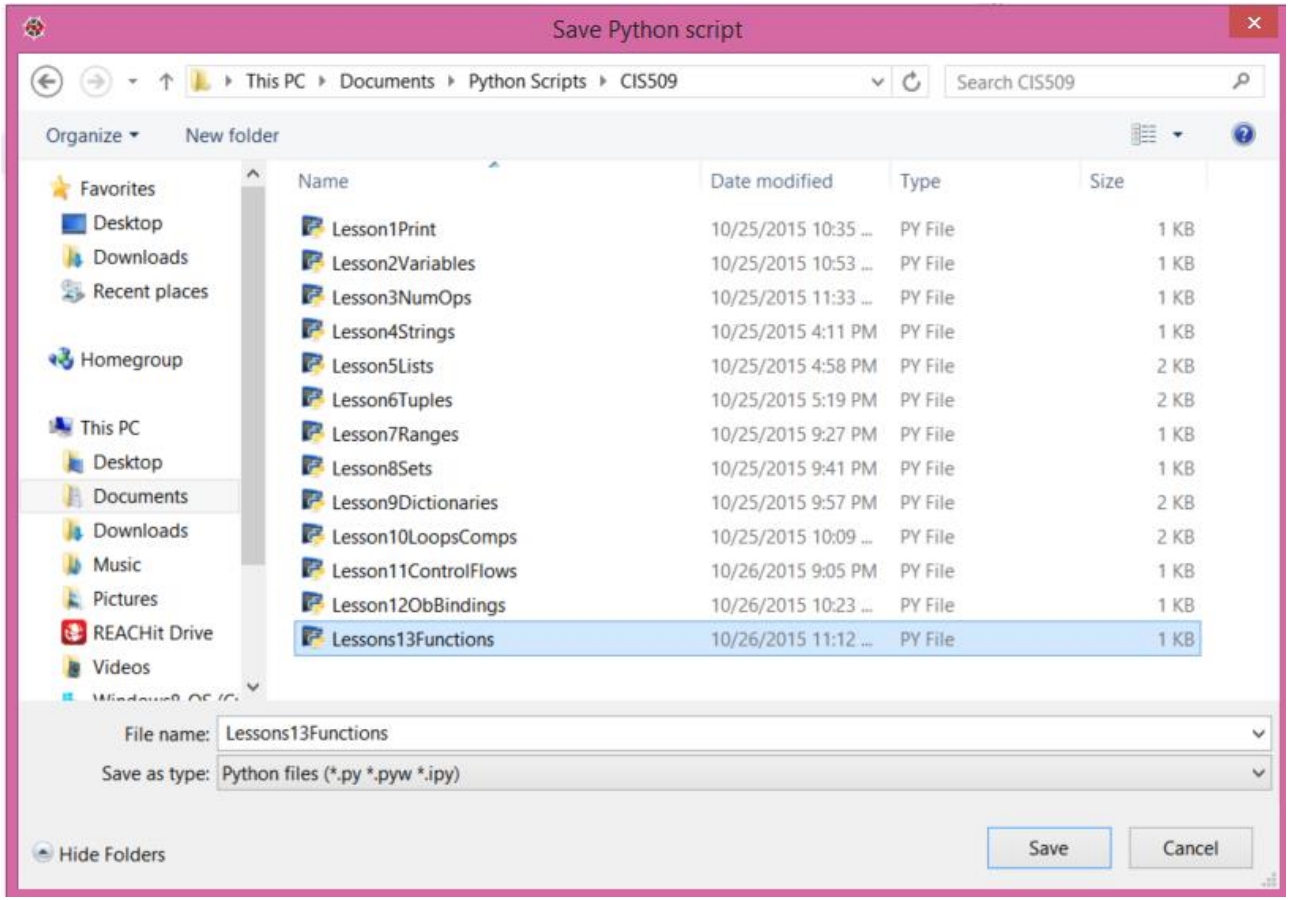
Functions

follow along!

(Reference: https://docs.python.org/3/tutorial/controlflow.html)

Create a new file called “Lesson13Functions.py”

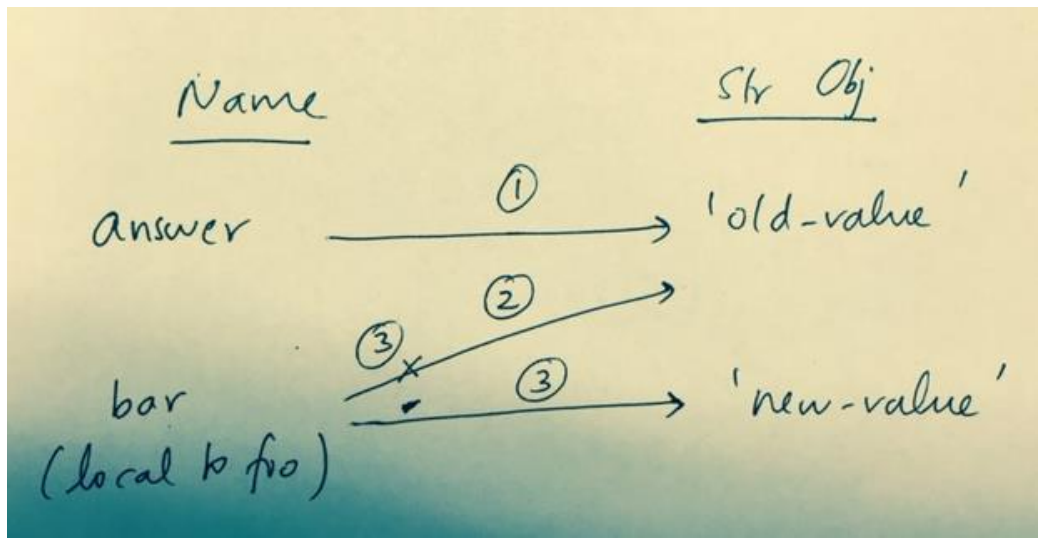
- Go to File -> New file…
- Go to File -> Save as…
- Got to CISPy directory
- Save file as “Lesson13Functions.py”



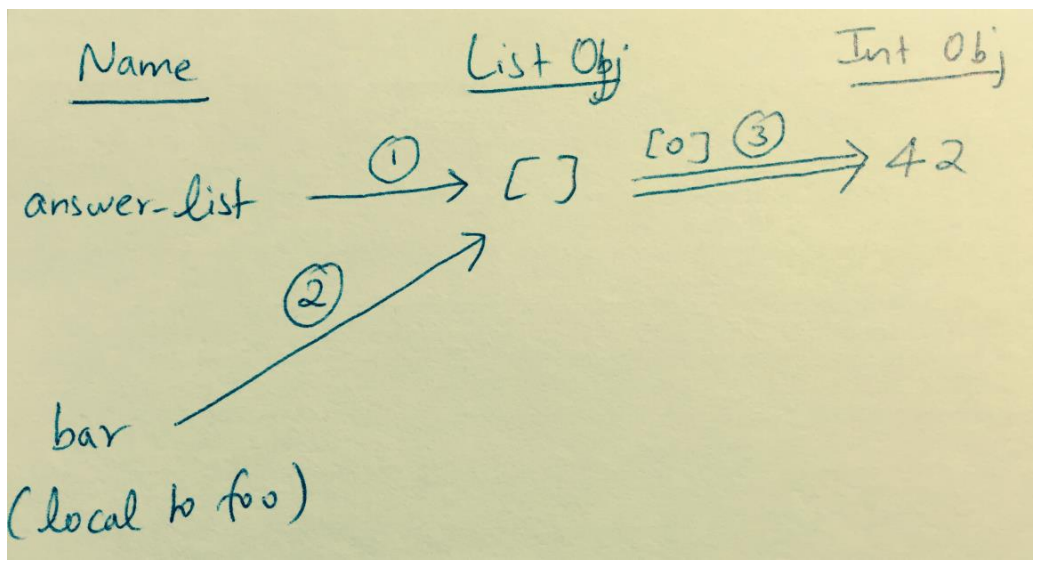
```python
# The keyword 'def' introduces a function definition.
# It must be followed by the function name and the parenthesized list of formal parameters.
# The statements that form the body of the function start at the next line, and must be indented.
def fib (n):
    # The first statement of the function body can optionally be a fucntion's
    # documentation string or docstring that summarizes what the function does
    """This function returns the Fibonacci Series."""
    result = []
    a, b = 0, 1
    for i in range(n):
        result.append(b)
        a, b = b, a+b
    return result

# print Document String
print (fib.__doc__)

# call function with arguments
print(fib(10))

print ()

# Parameters versus Arguments:
#     - Parameters are the variables in the function
#       (function fib(n) has parameter n)
#     - Arguments are the values given to the variables at the point of call
#       (function call fib(10) has argument 10)
#     - So outside the function, it is more common to talk about arguments.
#       Inside the function, you can really talk about either.
```

```
# Let's understand how function calls work in Python in terms of
# Objects, Bindings, and Scope

# Python is neither "call-by-reference" nor "call-by-value".
# In Python a variable is not an alias for a location in memory.
# Rather, it is simply a binding to a Python object.

# If I call foo(bar), I'm merely creating a binding within the scope of foo
# to the object that the argument bar is bound to when the function is called.
```

```
54 # If bar refers to an immutable object, the most that foo can do is create
55 # a name bar in its local namespace and bind it to some other object.
56 def foo (bar):
57
58     # (2) The Local Name bar is Bound to String Object 'old value'
59     print (bar)
60
61     # (3) The Local Name bar is now Bound to String Object 'new value'
62     #     (and no longer bound to String Object 'old value')
63     bar = 'new value'
64     print (bar)
65
66 # (1) The Name answer is Bound to the String Object 'old value'
67 answer = 'old value'
68 foo (answer)
69
70 # (4) The Name answer is still Bound to String Object 'old value'
71 print (answer)
```



```python
# If bar refers to a mutable object and foo changes its value,
# then these changes will be visible outside of the scope of the function.
def foo (bar):

    # (2) The Local Name bar is also now Bound to the List Object created
    print (bar)

    # (3) The 0th Element of the List Object created now contains the Int Object 42
    bar.append(42)
    print (bar)

# (1) The Name answer_list is Bound to the List Object []
answer_list = []

# (4) The Name answer_list is still Bound to the List Object
#     and reflects the changes made to it in the function call
foo (answer_list)
print (answer_list)
```



```
# The execution of a function introduces a new symbol table used for the local variables of the function.
# Variable references first look in the local symbol table, then in the local symbol tables of enclosing
# functions, then in the global symbol table, and finally in the table of built-in names.
# Thus, global variables cannot be directly assigned a value within a function (unless named in a global
# statement), although they may be referenced.

def demoFunc (arg1, arg2):

    # this will change local values of a1 and a2
    a1, a2 = -100, -200

    # this will change global value of a3
    global a3
    a3 = -300

    # this will pick up local copy of a1 and a2, and global copy of a3
    print ("demoFunc: ", a1, a2, a3)

# decalare global varaibles a1, a2 and a3
a1, a2, a3 = 10, 20, 30
print ("main: ", a1, a2, a3)

# function call will not change global copy of a1 and a2, but will change global value of a3
demoFunc (a1, a2)
print ("main: ", a1, a2, a3)

# this will change global copy of a1, a2 and a3
a1, a2, a3 = a1+5, a2+5, 35
print ("main: ", a1, a2, a3)

# function call will not change global copy of a1 and a2, but will change global value of a3
demoFunc (a1, a2)
print ("main: ", a1, a2, a3)
```

```
135
136 # It is  possible to define functions with a variable number of arguments.
137 # This can be done in three ways:
138 #     (1) Default argument values
139 #     (2) Keyword arguments
140 #     (3) Arbitrary argument lists
141
```

```
# (1) Default Argument Values

# You can specify a default value for one or more arguments
def isNumberInRange (num, i=25, n=100):
    if num in range(i, n):
        print ("found")
    else:
        print ("not found")
isNumberInRange(20)
isNumberInRange(20, 0)
isNumberInRange(20, 0, 10)
print ()

# Note: the default value is evaluated only once. This makes a difference when the default is a mutable object.
# So for instance this will print [1] on the first call, [1, 2] on the second, and [1, 2, 3] on the third.
def lstAppend(a, L = []):
    L.append(a)
    return L
print (lstAppend(1))
print (lstAppend(2))
print (lstAppend(3))
print()

# You can override this behavior as follows.
# This will print [1] on the first ccall, [2] on the second, and [3] on the third.
def lstAppend(a, L = None):
    if L is None:
        L = []
    L.append(a)
    return L
print (lstAppend(1))
print (lstAppend(2))
print (lstAppend(3))
print ()
```

```
177 # (2) Keyword Arguments
178
179 # Arguments can be of two kinds:
180 #     Keyword Argument:
181 #         An argument preceded by an identifier (e.g. name=) in a function call
182 #     Postional Argument:
183 #         An argument that is not a keyword argument
184
185 # In a function call, keyword arguments must follow positional arguments.
186 # All the keyword arguments passed must match one of the arguments accepted by the function,
187 # and their order is not important.
188 # No argument may receive a value more than once.
189
190 def func (arg1, arg2=0, arg3=-1, arg4=0):
191     print (arg1, arg2, arg3, arg4)
192
193 func (1000)                 # 1 positional argument
194 func (arg1=10)              # 1 keyword argument
195 func (arg1=10, arg2=100)    # 2 keyword arguments
196 func (arg2=100, arg1=10)    # 2 keyword arguments - order is not important
197 func (10, 100, 1000)        # 3 positional arguments
198 func (10, arg2=100)         # 1 positional and 1 keyword argument
199
200 # func (arg4=10, 100)       # this will yield non-keyword arg after keyword arg error
201 # func (arg5=10)            # this will yield unexpected keyword argument error
202 # func (0, arg1=0)          # this will yield multiple values for argument error
203
204 print()
```

```
206 # Keyword Argument can also passed as values in a dictionary preceded by **.
207 # Postional Argument can also be passed as elements of an iterable preceded by *.
208
209 def foo(*positional, **keywords):
210     print ("Positional:", positional)
211     print ("Keywords:", keywords)
212
213 # The *positional (tuple)  argument will store all of the positional arguments passed
214 # to foo(), with no limit to how many you can provide.
215 foo('one', 'two', 'three')
216
217 # The **keywords (dictionary) argument will store any keyword arguments:
218 foo(a='one', b='two', c='three')
219
220 # And of course, you can have both at the same time:
221 foo('one','two',c='three',d='four')
222 print()
```

```
# (3) Arbitrary Argument Lists

# You can also call a function with an arbitrary number of arguments.
# These arguments will be wrapped up in a tuple.
# Zero or more normal arguments may occur before the variable number of arguments.
# Any formal parameters which occur after the *args parameter can only be used
# as keyword (and not positional) arguments.

def concat (*args, sep='/'):
    print (sep.join(args))

concat ('john', 'jane')
concat ('john', 'jane', 'sandra')
concat ('john', 'jane', 'sandra', sep=',')

print()
```

What will these statements do?

```
# test

def func (lst):

    global a2
    a1 = 10

    lst.append(a1)
    lst.append(a2)

myList = []
print (myList)

a1, a2 = 5, 15
func(myList)
print (myList)

a1, a2 = 25, 35
func(myList)
print (myList)

print ()
```

```
def argListsCheck (argNorm1, argNorm2=10, *argPos, **argKey):
    print ("argNorm1 = ", argNorm1)
    print ("argNorm2 = ", argNorm2)
    print ("*argPos = ", argPos)
    print ("**argKey = ", argKey)

argListsCheck (0, 1, 2, 3, a=10, b=20, c=30)

print ()
```