Table of contents

---

1. Definition

(go to top)

In [1]:
```
def shout(word):
    """ Print string with three exclamation marks"""
    print(word + '!!!')
```

In [2]:
```
# Call Shout
shout('Python')
```
```
Python!!!
```

1.1. Functions with multiple arguments

(go to top)

In [4]:
```
def shout_3(word, word2):
    # Prints string with three exclamation marks
    print(word + word2 +  '!!!')
```

In [5]:
```
shout_3('Python', 'Rules')
```
```
PythonRules!!!
```

1.2. Python Builtins

(go to top)

In [35]:
```
import builtins

# Run this if you want to see
#dir(builtins)
```

---

2. Return Functions

(go to top)

In [7]:
```
def shout_2(word):
    # Returns string with three exclamation marks
    return word + '!!!'
```

In [8]:
```
yell = shout_2('Python')
print(yell)
print(yell, shout_2('Rules'))
```
```
Python!!!
Python!!! Rules!!!
```

2.1 Returns Multiple Values

(go to top)

In [9]:
```python
def shout_all(word1, word2):
    shout1 = word1 + '!!!'
    shout2 = word2 + '!!!'

    return shout1, shout2
```

In [10]:
```python
yell1, yell2 = shout_all('Python', 'Rules')
```

In [11]:
```python
print(yell1, yell2)
```

```
Python!!! Rules!!!
```

---

3. Arguments

(go to top)

3.1 Single default Argument

(go to top)

In [12]:
```python
# Define shout_echo

def shout_echo(word1, echo = 1):
    """Concatenate copies of word1 and three
     exclamation marks at the end of the string."""

    # Concatenate echo-copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Concatenate '!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!'

    # Return shout_word
    return shout_word
```

In [13]:
```
# Call shout_echo() with "Hey": no_echo
no_echo = shout_echo("Hey")

# Call shout_echo() with "Hey" and echo=5: with_echo
with_echo = shout_echo("Hey", 5)
```

In [14]:
```
# Print no_echo and with_echo
print(no_echo)
print(with_echo)
```

```
Hey!!!
HeyHeyHeyHeyHey!!!
```

3.2 Multiple default Arguments

(go to top)

In [15]:
```
# Define shout_echo
def shout_echo(word1, echo = 1, intense = False):
    """Concatenate copies of word1 and three exclamation marks at the end of 

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Make echo_word uppercase if intense is True
    if intense is True:
        # Make uppercase and concatenate '!!!': echo_word_new
        echo_word_new = echo_word.upper() + '!!!'
    else:
        # Concatenate '!!!' to echo_word: echo_word_new
        echo_word_new = echo_word + '!!!'

    # Return echo_word_new
    return echo_word_new
```

In [19]:
```
# Call shout_echo() with "Hey", echo=5 and intense=True: with_big_echo
with_big_echo = shout_echo("Hey", 5, True)

# Call shout_echo() with "Hey" and intense=True: big_no_echo
big_no_echo = shout_echo("Hey", intense = True)
```

In [20]:
```
# Print values
print(with_big_echo)
print(big_no_echo)
```

```
HEYHEYHEYHEYHEY!!!
HEY!!!
```

4. `*args` - Variable Length (Positional) Arguments

- (args OR anywordreally)

(go to top)

In [30]:
```
def find_type(*args):
    return type(args)

find_type("alpha", 'beta')
```

Out[30]:
```
tuple
```

In [29]:
```
# Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""

    # Initialize an empty string: hodgepodge
    hodgepodge = ""

    # Concatenate the strings in args
    for word in args:
        hodgepodge += word

    # Return hodgepodge
    return hodgepodge
```

In [30]:
```
# Call gibberish() with one string: one_word
one_word = gibberish('luke')

# Call gibberish() with five strings: many_words
many_words = gibberish("luke", "leia", "han", "obi", "darth")

# Print one_word and many_words
print(one_word)
print(many_words)
```

```
luke
lukeleiahanobidarth
```

5. `**kwargs` Variable Length Keyword Arguments

(go to top)

- can be **kwargs
- or **anythingreally
- what matters is the **

In [23]:
```
def find_type(**y):
    return type(y)

find_type(a = "alpha", b = 2)
```

Out[23]:
```
dict
```

In [41]:
```
def find_type(**y):
    for key, value in y.items():
        print(key + ": " , value)
        print(type(value))

find_type(a = "alpha", b = "2", c = 2)
```

```
a:  alpha
<class 'str'>
b:  2
<class 'str'>
c:  2
<class 'int'>
```

In [32]:
```
# Define report_status
def report_status(**kwargs):
    """Print out the status of a movie character."""

    print("\nBEGIN REPORT\n")

    # Iterate over the key-value pairs of kwargs
    for key, value in kwargs.items():
        # Print out the keys and values, separated by a colon ':'
        print(key + ": " + value)

    print("\nEND REPORT")
```

In [33]:
```
# First call to report_status()
report_status(name='luke', affiliation='jedi', status='missing' )

# Second call to report_status()
report_status(name='anakin' , affiliation='sith lord' , status='deceased' )
```

```
BEGIN REPORT

name: luke
affiliation: jedi
status: missing

END REPORT

BEGIN REPORT

name: anakin
affiliation: sith lord
status: deceased

END REPORT
```

---

6. Scope: Testing Scope

(go to top)

In [39]:
```
#global scope

new_val = 10
```

In [40]:
```
def square():
    new_val = 5 ** 2
    print(new_val, end=" ||  ")

square()
print(new_val)

# new_val unchanged in the global scope by the function square()
# new_val is accessible, global functions are accesible everywhere but cannot
# without global keyword
```

```
25 ||  10
```

---

7. Scope: `global` Keyword

- Access & change/affect object in the global scope inside a function

(go to top)

```
In [ ]: new_val = 10
```

```
In [41]: def square():

             global new_val

             new_val = new_val ** 2
             print(new_val, end=" ||  ")

         square()
         print(new_val)

         """ new_val IS ACCESSIBLE AND CHANGED in the global scope by the function squ
```

```
100 ||  100
```

```
Out[41]: ' new_val IS ACCESSIBLE AND CHANGED in the global scope by the function square
         '
```

```
In [44]: # Create a string: team
         team = "teen titans"
```

```
In [46]: # Define change_team()
         def change_team():
             """Change the value of the global variable team."""

             # Use team in global scope
             global team

             # Change the value of team in global: team
             team = "justice league"

             # Print team
             print(team, end="  ||   ")
```

In [47]:
```
# Call change_team()
change_team()

# Print team
print(team)

""" VALUE OF team CHANGES AFTER FUNCTION IS CALLED """
```

```
justice league  ||   justice league
```

Out[47]:
```
' VALUE OF team CHANGES AFTER FUNCTION IS CALLED '
```

---

8. Scope: `nonlocal` Keyword

- Acesss and affect an object in an outer function of nested loops

(go to top)

In [48]:
```
def outer():
    """Print n"""
    n = 1

    def inner():
        nonlocal n
        n = 4
        print(n)

    inner()
    print(n)
```

In [49]:
```
outer()
```

```
4
4
```

---

9. Nested Functions

(go to top)

In [2]:
```
# finds the k-root of n
def anyroot(n, k):
    """ Finds the k root of n """
    def root(n):
        return n ** (1/k)
    return root(n)
```

In [3]:
```
print(anyroot(4,2))
```

```
2.0
```

---

10. Nested Functions: Returns

(go to top)

In [1]:
```
# Define echo
def echo(n):
    """Returns inner function"""

    def inner_echo(word):
        """Concatenate copies or word"""
        return word * n

    return inner_echo
```

In [2]:
```
echo(2)('test')
```

Out[2]:
```
'testtest'
```

In [26]:
```
twice = echo(2) # repeats the word twice
thrice = echo(3) # repeats the word thrice

print(twice('hey you!'),  "||", thrice('hey there!'))
```

```
hey you!hey you! || hey there!hey there!hey there!
```

---

11. Nested Functions: Returns

(go to top)

In [11]:
```
# Define echo
def echo(n,word):
    """Returns inner function"""

    def inner_echo(n, word):
        """Concatenate copies or word"""
        return word * n

    return inner_echo(n, word)
```

In [21]:
```
print(echo(2, 'Python'))
```

```
PythonPython
```

In [22]:
```
print(echo(3, 'Python'))
```

```
PythonPythonPython
```

In [17]:
```
def raise_to(x, n):
    """Return x ^ n"""

    def inner(x):
        """ Raise x to the power of n"""
        raised = x ** n
        return raised

    return inner(x)
```

Out[17]: 8

In [23]:
```
raise_to(2,3)
```

Out[23]: 8

---

12. Lambda Function

- `lambda input: output`

(go to top)

In [7]:
```
raise_to_power = lambda x, y : x ** y
print(raise_to_power(2,4))
```

```
16
```

In [1]:
```
# Define echo_word as a lambda function
echo_word = (lambda word1, echo: word1 * echo)
echo_word('hey', 5)
```

Out[1]:
```
'heyheyheyheyhey'
```

In [37]:
```
f = lambda a,b: a if (a > b) else b
print(f(5,6))
```

```
6
```

---

13. Lambda Function: `map()`

- Takes a function and a sequence such as a list and applies the function over all elemets of the sequence
- map(function, sequence)

(go to top)

In [4]:
```
arr = map(int, input().split())
```

```
5 6 9 8
```

In [6]:
```
a = list(arr)
a
```

Out[6]:
```
[5, 6, 9, 8]
```

In [11]:
```
numbers = [48, 6, 9, 21, 1]

square_all = map(lambda num: num ** 2, numbers)

print(square_all)
print(list(square_all))
```

```
<map object at 0x111b24ed0>
[2304, 36, 81, 441, 1]
```

In [3]:
```
spells = ["protego", "accio", "expecto patronum", "legilimens"]

# Use map() to apply a lambda function over spells: shout_spells
shout_spells = map(lambda word: word + '!!!', spells)

# Print the result
print(list(shout_spells))
```

```
['protego!!!', 'accio!!!', 'expecto patronum!!!', 'legilimens!!!']
```

In [15]:
```python
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
temp = (36.5, 37, 37.5,39)

F = map(fahrenheit, temp)

print(list(F))
```

```
[97.7, 98.60000000000001, 99.5, 102.2]
```

In [4]:
```python
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'l

# Use filter() to apply a lambda function over fellowship: result
result_2 = map(lambda member: len(member) > 6 , fellowship)

# Convert result to a list: result_list
result_list = list(result_2)

# Print result_list
print(result_list)
```

```
[False, True, False, False, True, True, True, False, True]
```

---

14. Lambda Function: `filter()`

- The function filter() offers a way to filter out elements from a list that don't satisfy certain criteria.
- `filter(function, sequence)`

(go to top)

In [18]:
```python
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'l

# Use filter() to apply a lambda function over fellowship: result
result = filter(lambda member: len(member) > 6 , fellowship)

# Convert result to a list: result_list
result_list = list(result)

# Print result_list
print(result_list)
```

```
['samwise', 'aragorn', 'boromir', 'legolas', 'gandalf']
```

15. Lambda Function: `reduce()`

Definition

- The `reduce()` function is useful for performing some computation on a list
- Note that it returns the final cumulative not step-by-step result. i.e. it runs through whole sequence before giving an answer.
- It always takes 2 lambda parameters and, unlike `map()` and `filter()`, returns a single value as a result.

To use reduce(), you must import it from the functools module.

- The function `reduce(func, seq)` continually applies the function func() to the sequence seq. It returns a single value.
- If `seq = [ s1, s2, s3, ... , sn ]`, calling `reduce(func, seq)` works like this:
  - At first the first two elements of seq will be applied to func, i.e. `func(s1,s2)` The list on which `reduce()` works looks now like this: `[ func(s1, s2), s3, ... , sn ]`
  - In the next step func will be applied on the previous result and the third element of the list, i.e. `func(func(s1, s2),s3)`
  - The list looks like this now: `[ func(func(s1, s2),s3), ... , sn ]`
  - it will continue like this until just one element is left and return this element as the result of `reduce()`

(go to top)

In [8]:
```
# In this exercise, you will use reduce() and a lambda function that concaten

# Import reduce from functools
from functools import reduce

# Create a list of strings: stark
stark = ['B', 'sansa', 'arya', 'brandon', 'rickon']
```

In [9]:
```
# Use reduce() to apply a lambda function over stark: result
result = reduce(lambda child, child2: child, stark)
print(result)
```

```
B
```

In [12]:
```
result1 = reduce(lambda child, child2: child * 2 +'-', stark)
print(result1)
```

```
BB-BB--BB-BB---BB-BB--BB-BB----
```

In [29]:
```
result2 = reduce(lambda child, child2: child + child2, stark)
print(result2)
```

```
Bsansaaryabrandonrickon
```

In [32]:
```
print(reduce(lambda x,y: x+y, [47,11,42,13]))
```

```
113
```

In [35]:
```
f = lambda a,b: a if (a > b) else b
print(reduce(f, [47,11,42,102,13]))
```

```
102
```

In [34]:
```
print(reduce(lambda x, y: x+y, range(1,101)))
```

```
5050
```

---

16. Title

(go to top)

---

17. Title

(go to top)

---

18. Title

(go to top)

---

19. Title

(go to top)

---

20. Title

(go to top)

---

21. Examples

- Parameters are always passed by value. However, if the actual parameter is a variable whose value is a mutable object (like a list or graphics object), then changes to the state of the object will be visible to the calling program.
- The list is passed as a parameter and the change is visible (go to top)

In [10]:
```
def interest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1 + rate)
    print(balances)
```

In [11]:
```
def test():
    amounts = [1000,2000,3000,4000]
    rate = 0.05
    interest(amounts,rate)
    print(amounts)
```

In [12]:
```
test()
```

```
[1050.0, 2100.0, 3150.0, 4200.0]
[1050.0, 2100.0, 3150.0, 4200.0]
```

In [ ]:
```

```

In [17]:
```
def interest(balance, rate):
    balance = balance * (1 + rate)
    print(balance)
```

In [18]:
```
def test():
    amounts = 1000
    rate = 0.05
    interest(amounts,rate)
    print(amounts)
```

In [19]:
```
test()
```

```
1050.0
1000
```

In [ ]: