Functions and Scope

Recap

```python
def is_even(x):
    """
    Input: x, a positive integer
    Returns True if x is even, False otherwise
    """
    return x % 2 == 0
```

1. what is the function name?
2. where is the docstring?
3. what is the input?
4. what is the output?
5. where is the body of the function?
6. where is the return type?

Recap

what is this code doing:

```
x = is_even(54)
print(x)
```

Recap

what is this code doing:

```
x = is_even(54)
print(x)
```

Recap

what is this code doing:

```python
x = is_even(54)
print(x)
```

Recap

what is this code doing:

```python
x = is_even(54)
print(x)
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(1, 2):
    return 1 + 2
def mult(x, y):
    return x * y

add(1,2)
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

3
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```
def add(3, 4):
    return 3 + 4
def mult(x, y):
    return x * y

3
print(add(3,4))
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

3
print(7)
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```
def add(x, y):
    return x + y
def mult(3, 4):
    return 3 * 4

3
print(7)
mult(3,4)
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

3
print(7)
12
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(5, 6):
    return 5 * 6

3
print(7)
12
print(mult(5,6))
```

Recap

What is printed if you run this code?

```python
def add(x, y):
    return x + y
def mult(x, y):
    return x * y

3
print(7)
12
print(30)
```

Return

- return only has meaning **inside** a function
- only one return executed inside a function
- code inside function after return is not executed
- has a value associated with it, given to the function caller

Print

- can be used **outside** functions
- has a value associated with it, outputted to the console
- print expression itself returns `None` value

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) ≥ epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) ≥ epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) ≥ epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

```python
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low) / 2.0

    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low) / 2.0
    return ans
```

Functions as a support of modularity

So to get the square root of 25, we can just call:

```
print(bisection_root(25))
```

This is now a **black box**, with a specified input, and an output that changes based on the input

Zooming out and scope

```python
def sum_odd(a, b):
    sum_of_odds = 0

    for i in range(a, b+1):
        if i % 2 == 1:
            sum_of_odds += i

    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Variables

sum_odd	function object
low	2
high	7
my_sum	

Zooming out and scope

```python
def sum_odd(a, b):
    sum_of_odds = 0

    for i in range(a, b+1):
        if i % 2 == 1:
            sum_of_odds += i

    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Variables

sum_odd	function object
low	2
high	7
my_sum	

Zooming out and scope

```python
def sum_odd(a, b):
    sum_of_odds = 0

    for i in range(a, b+1):
        if i % 2 == 1:
            sum_of_odds += i

    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Variables

sum_odd	function object
low	2
high	7
my_sum	

Zooming out and scope

```python
def sum_odd(a, b):
    sum_of_odds = 0

    for i in range(a, b+1):
        if i % 2 == 1:
            sum_of_odds += i

    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Variables

sum_odd	function object
low	2
high	7
my_sum	

Zooming out and scope

```python
def sum_odd(a, b):
    sum_of_odds = 0

    for i in range(a, b+1):
        if i % 2 == 1:
            sum_of_odds += i

    return sum_of_odds

low = 2
high = 7
my_sum = sum_odd(low, high)
```

Variables

sum_odd	function object
low	2
high	7
my_sum	15

Function scope

How does python execute a function

Python creates an **entirely new environment** with every function call

- essentially a *mini program*
- it runs with its *own set* of variables
- it does the work (body)
- it returns a value
- then the environment is destroyed

Environments

- Global environment
  - created when program starts
  - holds all global variables
  - holds function definitions

And invoking a function creates a **new** environment (scope)

Variable Scope

- *formal parameters/arguments* get bound to the value of *input parameters*
- *scope* is a mapping of names to objects
  - it defines which variables have which values
- expressions in bodies of a function evaluate with respect to the function’s scope

```python
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

Variable Scope

- *formal parameters/arguments* get bound to the value of *input parameters*
- *scope* is a mapping of names to objects
    - it defines which variables have which values
- expressions in bodies of a function evaluate with respect to the function's scope

```python
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

Variable Scope

- *formal parameters/arguments* get bound to the value of *input parameters*
- *scope* is a mapping of names to objects
  - it defines which variables have which values
- expressions in bodies of a function evaluate with respect to the function’s scope

```python
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

Variable Scope

- *formal parameters/arguments* get bound to the value of *input parameters*
- *scope* is a mapping of names to objects
  - it defines which variables have which values
- expressions in bodies of a function evaluate with respect to the function's scope

```python
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

Variable Scope

- *formal parameters/arguments* get bound to the value of *input parameters*
- *scope* is a mapping of names to objects
  - it defines which variables have which values
- expressions in bodies of a function evaluate with respect to the function's scope

```python
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

After line 3

```
1  def f(x):
2      x = x + 1
3      return x
4
5  x = 3
6  z = f(x)
```

Global Scope

f	function object

After line 5

```
1  def f(x):
2      x = x + 1
3      return x
4
5  x = 3
6  z = f(x)
```

Global Scope

f	function object
x	3

During line 6

```
def f(x):
    x = x + 1
    return x

x = 3
z = f(x)
```

Global Scope

f	function object
x	x

During line 6

```
1  def f(x):
2      x = x + 1
3      return x
4
5  x = 3
6  z = f(x)
```

Global Scope

f	function object
x	3

Function Scope (inside f)

x	x = 3

During line 6

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3

Function Scope (inside f)

x	3

During line 6

```
1   def f(x):
2       x = x + 1
3       return x
4
5   y = 3
6   z = f(y)
```

Global Scope

f	function object
y	3

Function Scope (inside f)

x	3

inside the function

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3

Function Scope (inside f)

x	3

inside the function

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3

Function Scope (inside f)

x	3

inside the function

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3

Function Scope (inside f)

x	4

back outside the function

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3
z	4

Function Scope (inside f)

- `f(y)` gets replaced by return value `4`
- the scope gets destroyed

x	4

Variable scope

```python
def f(x):
    x = x + 1
    return x

y = 3
z = f(y)
```

Global Scope

f	function object
y	3
z	4

Other examples

You can access a variable defined outside the function

```python
def g(y):
    print(x)
    print(x + 1)
x = 5
g(x)
print(x)
```

But you can't change it

```python
def h(y):
    x += 1
x = 5
h(x)
print(x)
```

Higher order procedures

- Objects in Python have a type
  - int, float, str, boolean, NoneType, function
- objects can appear in the right hand side of assignment statement
  - bind a name to an object
- Objects
  - can be used as an argument to a procedure/function
  - can be returned as a value from a procedure
- Functions are _____

We can treat functions just like any other object

- can be arguments to another function
- can be returned by another function

Function as an object

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Program scope

calc	function object
add	function object
div	function object
res	

Create calc scope

```
def calc(op, x, y):
    return op(x, y)

res = calc(add, 2, 3)
```

Prog Scope		Calc Scope	
calc	function object	op	Add
add	function object	x	2
div	function object	y	3
res			

Run the calc body

```python
def calc(op, x, y):
    return op(x, y)
```

Run the calc body

```python
def calc(add, 2, 4):
    return add(2, 4)
```

Define add scope

```python
def calc(add, 2, 4):
    return add(2, 4)

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

Prog Scope		Calc Scope		Add Scope	
calc	function object	op	Add	a	2
add	function object	x	2	b	4
div	function object	y	3		
res					

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope

calc	function object
add	function object
div	function object
res	6

Calc Scope

op	Add
x	2
y	3

Add Scope

a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return a + b

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return 2 + 4

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return op(x, y)

def add(a, b):
    return 6

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope

calc	function object
add	function object
div	function object
res	6

Calc Scope

op	Add
x	2
y	3

Add Scope

a	2
b	4

Evaluating

```python
def calc(op, x, y):
    return 6

def add(a, b):
    return 6

def div(a, b):
    if b != 0:
        return a/b
    print("division by zero!")

res = calc(add, 2, 3)
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Evaluating

```
def calc(op, x, y):
    return 6

def add(a, b):
    return 6

def div(a, b):
    if b ≠ 0:
        return a/b
    print("division by zero!")

res = 6
```

Prog Scope	
calc	function object
add	function object
div	function object
res	6

Calc Scope	
op	Add
x	2
y	3

Add Scope	
a	2
b	4

Exercise

Trace this function call

```python
def calc(op, x, y):
    return op(x, y)

def div(a, b):
    if b != 0:
        return a/b
    print("division by zero!")

res = calc(div, 2, 0)
```

What is the value of `res` and what get's printed?

make global scope table, and function scope tables

More Exercise

```python
1   def func_a():
2       print("inside func_a")
3
4   def func_b(y):
5       print("inside func_b")
6       return y
7
8   def func_c(f, z):
9       print("inside func_c")
10      return f(z)
11
12  print(func_a())
13  print(5 + func_b(2))
14  print(func_c(func_b, 3))
```

Starting at line 12

Exercise

Write a function that meets these specs

```python
def apply(criteria, n):
    """
    criteria, a function that takes in a number and returns a boolean
    n, an integer

    Returns how many ints from 0 to n (inclusive) meet the criteria
    (i.e. numbers that return True when a criteria is applied)
    """

def is_even(x):
    """
    x, a positive integer
    returns True if x is even, False otherwise
    """
```

Dictionaries

key value stores, hash maps

Dictionary

A dictionary is a **key-value** store. It's primary use case is

1. Storing multiple pieces of data
2. Accessing data with keys instead of indices

A dictionary is *mutable* and *unordered*.

- In python it's defined using curly braces `{}`
- Where each key is separated from its value by a colon `:`
- and each key-value pair is separated by a comma `,`

```python
my_dictionary = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
```

Example problem

What if we wanted to store the health of different enemies in the game

With lists one way of doing this is

```python
enemies = ["goblin", "troll", "dragon"]
```

```python
health = [30, 50, 100]
```

How would you access the health of the troll?

```python
for i in range(len(___)):
    if ___[i] == "___":
        print(___[i])
```

Example problem

What if we wanted to add a new value for `"orc"` with health `40`

```python
enemies.append("orc")
health.append(40)
```

And if wanted to delete the `"goblin"`

```python
index = enemies.index("goblin")
enemies.pop(index)
health.pop(index)
```

Exercise

Given

```python
enemies = ["goblin", "troll", "dragon"]
weight = [30, 50, 100]
```

Write a function where

```python
def get_weight(enemy_name):
    """
    input: enemy_name (String)
    return: weight (int)
    """
```

Dictionary solution

```python
enemy_health = {
    "goblin": 30,
    "troll": 50,
    "dragon": 100
}
```

```python
enemy_health["troll"]
```

Hashing

Dictionaries work by using a process called *hashing*

Recall that lists are stored in **straight** blocks of memory. That is why lists are ordered, and why indices start at 0

A dictionary uses a **hash function** to convert the key into a memory address

And when a key is looked up, the hash function is used again to find the memory address

Example

Assume a key `"health"` is hashed, and assume the has function takes the number of characters, and multiplies it by the index of the last character

- This would mean that `"health"` would be hashed to `6 * 5 = 30`
- and `"mana"` would be hashed to `_ * _ = __`

Adding stamina would put it in the memory address `__`

This means that if you use `enemy_stats["stamina"]`, under the hood, it also does the hash function, and goes to that memory address

Exercise 2

Given the dictionary

```python
player = {
    "name": "Hero",
    "level": 5,
    "health": 100,
    "mana": 50
}
```

complete the following functions

```python
def level_up():
    """
    increases the player's level by 1
    input: None
    output: None
    """
```

```python
def take_damage(damage):
    """
    decreases the player's health by damage
    input: damage (int)
    output: None
    """

def use_mana(cost):
    """
    decreases the player's
        mana by cost
    input: cost (int)
    output: None
    """

def is_alive():
    """
    checks if the player
        is alive (health > 0)
    input: None
    return: (bool)
    """
```