Functions

CS195 - Lecture 10
Instructor: Dr. V

Lecture 10

- why use functions
- def, return
- arguments
  - passing args
  - default args
- variable scope
  - local vs global variables
- docstring

repeating blocks of code

```
# in your RPG you had code like this:

...
print('You found an item! Do you want to pick it up?')
if not input('(Y/n) ').lower() == 'n':
    ...
...
print('Monster attack! Do you want to fight back?')
if not input('(Y/n) ').lower() == 'n':
    ...
...
print('Do you want to fight the creature?')
if not input('(Y/n) ').lower() == 'n':
    ...
...
```

repeating blocks of code

```
# what if you wanted to change this to make NO the default?
#   ugh...
...
print('You found an item! Do you want to pick it up?')
if not input('(y/N) ').lower() == 'y':
    ...
...
print('Monster attack! Do you want to fight back?')
if not input('(y/N) ').lower() == 'y':
    ...
...
print('Do you want to fight the creature?')
if not input('(y/N) ').lower() == 'y':
    ...
...
```

repeating blocks of code

```
# now imagine you wrote a confirm(prompt) function:
def confirm(prompt):
    print(prompt)
    return not input('(Y/n) ').lower() == 'n'

...
if confirm('You found an item! Do you want to pick it up?'):
    ...
...
if confirm('Monster attack! Do you want to fight back?'):
    ...
...
if confirm('Do you want to fight the creature?'):
    ...
...
```

repeating blocks of code

```
# what if you wanted to change this to make NO the default:
def confirm(prompt):
    print(prompt)
    return not input('(y/N) ').lower() == 'y'

...
if confirm('You found an item! Do you want to pick it up?'):
    ...
...
if confirm('Monster attack! Do you want to fight back?'):
    ...
...
if confirm('Do you want to fight the creature?'):
    ...
...
```

Functions are better

- rather than repeating lengthy blocks of code, use functions
  - less code, less copying/pasting
  - better readability
  - centralized logic
    - if you need to change what this block of code does, you only need to change it in that one function, rather than in every single place where that code is executed

python functions

- a function in python
  - is a named block of code
  - can take in arguments
  - always returns a value (by default returns None)

```
def greeting():          # define a function
    print("hello")


...
greeting()               # execute the function
```

python functions

- a function in python
  - is a named block of code
  - can take in arguments
  - always returns a value (by default returns None)

```
def greeting():          # define a function
    print("hello")
    return 1
...
x = greeting()           # execute the function
```

python functions

```
1  def hello():
2      return 'hello'
3
4  def goodbye():
5      print('goodbye')
6
7  #what does this output?
8  print( hello() )
9
10 #what does this output?
11 print( goodbye() )
12
13
14
15
```

python functions - passing arguments

```
1  def square(x):
2      return x**2
3
4  z = square(2)
5  z = square(z)
6
7  #what does this output?
8  print( square(z) )
9
10
11
12
13
14
15
```

python functions - passing arguments

```
1  def exp(x, y):
2      return x**y
3
4  z = exp(2, 4)
5  z = exp(z, 2)
6
7  #what does this output?
8  print( z )
9
10
11
12
13
14
15
```

python functions - default argument values

```
1  def exp(x, y=2):
2      return x**y
3
4
5  #what does this output?
6  print( exp(2, 4) )
7
8  #what does this output?
9  print( exp(2) )
10
11
12
13
14
15
```

functions calling other functions

```
1  def addSquare(x,y):
2      x **= 2
3      print(f'adding {x}')
4      return x+y
5
6  def sumOfSquares(l):
7      total = 0
8      for val in l:
9          total = addSquare(val, total)
10     return total
11
12 l = range(1,5)
13 print( f'the total is {sumOfSquares(l)}' )
14
15
```

python functions - variable scope

```
1  def square(x):
2      y = x**2       # these x and y do not exist outside this def
3      return y
4
5  x = square(2)      # these x and y are *NOT* the same vars as above
6  y = square(x)
7
8  #what does this output?
9  print( y+x )
10
11
12
13
14
15
```

Local vs Global variables

- variables defined outside function definitions are global – visible by all functions

python functions - variable scope

```
1  x = 200
2
3  def foo():
4      print(x)
5
6  # what does this print?
7  foo()
8
9
10
11
12
13
14
15
```

Local vs Global variables

- variables defined outside function definitions are global – visible by all functions
- but...
  - if a variable looks like it's being defined inside a function, it is only visible to that function

python functions - variable scope

```
1  x = 200       # globally-defined x
2
3  def foo():
4      x = 300   # locally-defined x
5      print(x)
6
7  # what does this print?
8  foo()
9
10 # what does this print?
11 print(x)
12
13
14
15
```

Local vs Global variables

- variables defined outside function definitions are global – visible by all functions
- but...
  - if a variable looks like it's being defined inside a function, it is only visible to that function
  - but...
    - if you use a global keyword, you can tell the function explicitly that a certain variable is global

python functions - variable scope

```
1  x = 200       # globally-defined x
2
3  def foo():
4      global x
5      x = 300  # same x that was defined on line 1
6      print(x)
7
8  # what does this print?
9  foo()
10
11 # what does this print?
12 print(x)
13
14
15
```

python functions - variable scope

```
1  x = 200
2
3  def foo():
4      x = 300
5      def bar():
6          x = 400
7          print(x)
8      bar()
9      print(x)
10
11 # what does each of these print?
12 foo()
13 print(x)
14
15
```

python functions - variable scope

```
1  x = 200
2
3  def foo():
4      x = 300
5      def bar():
6          x = 400
7          print(x)
8
9      print(x)
10
11 # what does each of these print?
12 foo()
13 print(x)
14
15
```

python functions - variable scope

```
1  x = 200
2
3  def foo():
4      x = 300
5      def bar():
6          x = 400
7          print(x)
8
9      print(x)
10
11 # what does each of these print?
12 foo()
13 print(x)
14 bar()
15
```

python docstring

```
1  # add triple-quoted (indented) text atop your function code-block
2  #   to document what your function does
3
4  def power(x, y): # single-line docstring example
5      """Returns x raised to power y."""
6      return x**y
7
8
9  def greeting():  # multi-line docstring example
10     """
11     Prints a greeting.
12     Also does some other stuff.
13     """
14     print("Hello!")
15     ...
```

python docstring

```
1  def add_binary(a, b):
2      '''Returns the sum of two decimal numbers in binary digits.
3
4              Parameters:
5                      a (int): A decimal integer
6                      b (int): Another decimal integer
7
8              Returns:
9                      binary_sum (str): Binary string of the sum of
10                                       a and b
11     '''
12     binary_sum = bin(a+b)[2:]
13     return binary_sum
14
15
```

Assignment 9

- create a jupyter notebook a9.*yourLastName*.ipynb
  - add a markdown block atop the notebook with your name, class number/section, assignment number
  - add the following python code blocks
    - create a function (with docstring) called greeting that takes in one argument called name; function should
      - print(f"Hello, {name}")
      - return length of name
    - create a function (with docstring) called greetPeople that takes in one argument called names; function should
      - use a for-loop to call greeting for every name in names
      - return sum of all values returned by greeting
    - greetPeople(["Rashi","JJ","Nina","Indira"])

Assignment 9 - expected output

- expected output:

```
Hello, Rashi
Hello, JJ
Hello, Nina
Hello, Indira

17
```

Assignment 9 - Extra Credit

- write the greetPeople functions using just two lines of code
  - line 1: def ...
  - line 2:   return ...