Advanced Concepts in Python Functions

CS195 - Lecture 11
Instructor: Dr. V

Lecture 11

- type specification
- assert
- early return and guard clauses
- arguments
  - positional vs keyword args
  - *, **
- destructive functions
- function names are variables too (in python)
- decorators

argument types and assert

documenting your functions

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

documenting - with argument and return types

Specific way to replace docstring, better documentation

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

Specify types for arguments and for what's returned

documenting - with argument and return types

```
1  def mean(x):
2      return sum(x)/len(x)
3
4  def mse(x, y):
5      return mean([(a-b)**2 for a,b in zip(x,y)])
6
7
8
9
10
11
12
13
14
15
```

documenting - with argument and return types

```
1  def mean(x:list[float]) -> float:
2      '''Returns the mean value in list of values.'''
3      return sum(x)/len(x)
4
5  def mse(x:list[float], y:list[float]) -> float:
6      '''Mean squared error
7
8      Returns the mean squared difference between all
9      respective pairs of values in two lists.
10
11     (https://en.wikipedia.org/wiki/Mean_squared_error)
12     '''
13     return mean([(a-b)**2 for a,b in zip(x,y)])
14
15
```

Function arg and return types

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

- Indeed, you can specify what the argument and return types are in Python, but...
  - this is ONLY for documentation purposes
  - there will be NO runtime errors if you pass the wrong types of arguments to such functions
- So how do you ensure errors are thrown if wrong types are being passed?

Python assert statement

- use python assert statement to ensure some condition at any point in your code; i.e.,

```
assert condition[, assertion_message]
```

throws an error if the condition is false

- if the condition is False, your code will throw an error (and display whatever `assertion_message` you specified)

python assert

```
1  # add assert atop your function to throw errors whenever
2  #  variables don't fit the conditions you need them to satisfy
3
4  def sse(x, y):
5      '''Sum of squared error'''
6      assert type(x)==type(y)==list,
7                  "arguments must be lists of numbers"
8      return sum([(a-b)**2 for a,b in zip(x,y)])
9
10 sse(2,3)
11
12 AssertionError: arguments must be lists of numbers
13
14
15
```

python assert

```
1  # you can have multiple assertions
2  #  and they can be anywhere in your code, not just atop functions
3
4  def sse(x, y):
5      '''Sum of squared error'''
6      assert type(x)==type(y)==list,
7                  "arguments must be lists of numbers"
8      assert len(x) == len(y), "lists must be of equal length"
9      return sum([(a-b)**2 for a,b in zip(x,y)])
10
11 sse([2,2.1,3,4.5],[3,4,5])
12
13 AssertionError: lists must be of equal length
14
15
```

python assert

```
1  # assert isn't just for checking types,
2  #  you can use assert to make sure of *ANY* condition
3
4  def rating(item, r:int):
5      assert isinstance(r,int), "Rating must be an integer."
6      assert 0<r<=5, "Rating must be between 1 and 5."
7      print(f'Your rating for {item} is {r}.')
8
9  rating('Cowboy Bebop', 10)
10
11 AssertionError: Rating must be between 1 and 5.
12
13
14
15
```

condition to check

early returns

```python
def includesOdd(lst:list[int]) -> bool:
    "Returns True if lst has odd numbers, otherwise returns False"
    hasOdd = False
    assert isinstance(lst,list), "lst must be a list of integers"
    for num in lst:
        if num%2: # check if num is odd
            hasOdd = True
    return hasOdd

# what does each of these statements print?
print( includesOdd([2,4,6,8,10]) )
print( includesOdd([1,4,5,8,10]) )

# can we make the code above more efficient?

```

assert early, don't waste compute time

```
1  def includesOdd(lst:list[int]) -> bool:
2      "Returns True if lst has odd numbers, otherwise returns False"
3      assert isinstance(lst,list), "lst must be a list of integers"
4      hasOdd = False
5      for num in lst:
6          if num%2: # check if num is odd
7              hasOdd = True
8      return hasOdd
9
10 # what does each of these statements print?
11 print( includesOdd([2,4,6,8,10]) )
12 print( includesOdd([1,4,5,8,10]) )
13
14 # can we make the code above even more efficient?
15
```

break early, don't waste compute time

```
1  def includesOdd(lst:list[int]) -> bool:
2      "Returns True if lst has odd numbers, otherwise returns False"
3      assert isinstance(lst,list), "lst must be a list of integers"
4      hasOdd = False
5      for num in lst:
6          if num%2: # check if num is odd
7              hasOdd = True
8              break
9      return hasOdd
10
11 # what does each of these statements print?
12 print( includesOdd([2,4,6,8,10]) )
13 print( includesOdd([1,4,5,8,10]) )
14
15 # can we make the code above even more efficient?
```

return early, don't waste compute time

```
1  def includesOdd(lst:list[int]) -> bool:
2      "Returns True if lst has odd numbers, otherwise returns False"
3      assert isinstance(lst,list), "lst must be a list of integers"
4      for num in lst:
5          if num%2: # check if num is odd
6              return True
7      return False
8
9  # what does each of these statements print?
10 print( includesOdd([2,4,6,8,10]) )
11 print( includesOdd([1,4,5,8,10]) )
12
13
14
15
```

return early, don't waste compute time

```
1  def includesOdd(lst:list[int]) -> bool:
2      "Returns True if lst has odd numbers, otherwise returns None"
3      assert isinstance(lst,list), "lst must be a list of integers"
4      for num in lst:
5          if num%2: # check if num is odd
6              return True
7
8
9  # what does each of these statements print?
10 print( includesOdd([2,4,6,8,10]) )
11 print( includesOdd([1,4,5,8,10]) )
12
13
14
15
```

```python
def includesNoOdds(lst:list[int]) -> bool:
    """Returns True only if lst has no odd numbers."""
    assert isinstance(lst,list), "lst must be a list of integers"
    for num in lst:
        if num%2:
            return
    return True

# what does each of these statements print?
print( includesNoOdds([2,4,6,8,10]) )
print( includesNoOdds([2,4,5,8,10]) )
```

Early returns

- An early `return` is much like a `break` statement in a loop
- It is often recommended that you use "guard clauses", where you check for possible return conditions atop your functions

```
def foo(x,y):
    if iDontLike(x):   # guard clause
        return
    doSomethingWith(y)
    …

# note - there was no need for else
```

```
def foo(x,y):
    if not iDontLike(x):
        doSomethingWith(y)
        …
```

positional args, named args

positional arguments

```
1  def foo(x,y,z):
2      return x**3+y**2+z
3
4  print( foo(3,4,5) )
5
6
7
8
9
10
11
12
13
14
15
```

positional vs keyword arguments

```
1  def foo(x,y,z):
2      return x**3+y**2+z
3
4  print( foo(3,4,5) )
5
6  # same thing as above, but specifying args in any order
7  print( foo(y=4,z=5,x=3) )
8
9
10
11
12
13
14
15
```

keyword arguments

```
1  def foo(x=0,y=0,z=0):
2      return x**3+y**2+z
3
4  print( foo(y=4) )
5
6
7
8
9
10
11
12
13
14
15
```

mixing positional and keyword args

```
1  def foo(x,y=0,z=0):
2      return x**3+y**2+z
3
4  # you can specify arguments by position,
5  #   and if you do, they must come BEFORE keyword args
6  print( foo(3,z=4) )
7
8  # or, equivalently, just specify them all by keyword
9  print( foo(x=3,z=4) )
10
11 # regardless, if an argument doesn't have a default value,
12 #  it MUST be specified when function is called
13
14
15
```

positional arguments

```
1
2  # default args always follow non-default ones
3  def foo(x,y=0,z=0):
4      return x**3+y**2+z
5
6
7
8
9  # CANNOT have non-default args after default ones
10 def foo(x,y=0,z):
11     return x**3+y**2+z
12
13 SyntaxError: non-default argument follows default argument
14
15
```

*args and **kwargs

positional arguments

```
1  def adder(x,y,z):
2      print("sum:",x+y+z)
3
4  # what does this print?
5  adder(3,4,5)
6
7  # what does this print?
8  adder(5,10,15,20,25)
9
10
11
12
13
14
15
```

unlimited positional arguments

```
1  def adder( *values ):
2      # values becomes a tuple of all args passed to this function
3      total = 0
4      for val in values:
5          total += val
6      print("sum:",total)
7
8  # what does this print?
9  adder(3,4,5)
10
11 # what does this print?
12 adder(5,10,15,20,25)
13
14
15
```

split list into positional arguments

```
1  def foo(x,y=0,z=0):
2      print( x**3+y**2+z )
3
4  # what if you had a list you wanted to split into positional args
5  l = [3,4,5]
6
7  # you can do it manually, e.g.:
8  adder(l[0], l[1], l[2])
9
10 # but using * before list-name would do it for you automatically
11 adder( *l )
12
13 # the * operator would work for splitting any iterable into args
14 adder( *range(3,6) )
15
```

split list into positional arguments

```
1  # print() is a function that allows unlimited args
2  #    signature of print looks like this:
3  #  print(*values, end='\n', sep=' ', file=sys.stdout, flush=False)
4
5
6  l = [3,4,5]
7
8  # what does this print?
9  print( l )
10
11 # what does this print?
12 print( *l )
13
14
15
```

keyword arguments

```
1  def printProfile(name,age,gender):
2      print('-'*40)
3      print(f"name: {name}")
4      print(f"age: {age}")
5      print(f"gender: {gender}")
6      print('-'*40)
7
8  # what does this print?
9  printProfile(age=22,name='sandra',gender='f')
10
11 # what does this print?
12 printProfile(age=22,gender='m',name='joejoe',
13                city='troy',state='ny')
14
15
```

unlimited keyword arguments

```
1  def printProfile(**profile):
2      print('-'*40)
3      for k,v in profile.items():
4          print(f'{k}: {v}')
5      print('-'*40)
6
7
8  # what does this print?
9  printProfile(age=22,name='sandra',gender='f')
10
11 # what does this print?
12 printProfile(age=21,gender='m',name='joejoe',
13                city='troy',state='ny')
14
15
```

split dict into keyword arguments

```
1  def printProfile(name,age):
2      print( f'name: {name}; age: {age}' )
3
4  # what if you had a dict you wanted to split into keyword args
5  d = {'name':'jj','age':22}
6
7  # you can do it manually, e.g.:
8  printProfile( name = d['name'], age = d['age'] )
9
10 # but using **d would do it for you automatically
11 printProfile( **d )
12
13
14
15
```

mixing positional and keyword args

```
1  def foo(x=0, *args, y=8, **kwargs):
2      print(x)
3      print(args)
4      print(kwargs)
5      print(y)
6
7  # what does this print?
8  foo()
9  # what does this print?
10 foo(1,2,3,4)
11 # what does this print?
12 foo(1,2,3,4,a=5,b=6,c=7)
13 # what does this print?
14 foo(a=5,b=6,c=7,x=8,y=9)
15
```

destructive functions

python functions - changing values

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

python functions - changing values

```
1  def foo(l):
2      l[0]+=1
3      print(l[0])
4
5  l=[10,11,12]
6
7  #what does this print?
8  foo(l)
9
10 #what does this print?
11 print(l[0])
12
13
14
15
```

python functions - changing values

```
1  def foo(l): #this function is destructive. how do we fix this?
2      l[0]+=1
3      print(l[0])
4
5  l=[10,11,12]
6
7  #what does this print?
8  foo(l)
9
10 #what does this print?
11 print(l[0])
12
13
14
15
```

python functions - changing values

```
1  def foo(l):
2      l=l.copy()
3      l[0]+=1
4      print(l[0])
5
6  l=[10,11,12]
7
8  #what does this print?
9  foo(l)
10
11 #what does this print?
12 print(l[0])
13
14
15
```

python functions - changing values

```
1  def foo(l):
2      x = l[0] + 1
3      print(x)
4
5  l=[10,11,12]
6
7  #what does this print?
8  foo(l)
9
10 #what does this print?
11 print(l[0])
12
13
14
15
```

functions are variable too

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 bar()
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()        # how would you fix this so that bar() can run here?
15
```

python functions - variable scope

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

python functions - functions are variables too

```
1
2  def foo():
3      global z
4      def bar():
5          x = 400
6          print(x)
7      z = bar
8
9  foo()
10
11 z()
12
13
14
15
```

python functions - returning functions

```
1
2  def foo():
3      def bar():
4          x = 400
5          print(x)
6      return bar    # yes, a function can return a function
7
8  z = foo()
9
10 z()
11
12
13
14
15
```

python closures

```
1  # a closure is a combination of a function bundled together
2  #  (enclosed) with references to its surrounding state
3
4  def createMultiplier(x):
5      x *= 1000
6      def f(y):
7          print(y*x)
8      return f
9
10 m2 = createMultiplier(2)
11 m4 = createMultiplier(4)
12
13 # what do these print?
14 m2(3)
15 m4(3)
```

python decorators

```
1  # a decorator is a function that takes takes another function
2  #  as an argument, and returns some variation of that function
3  def my_decorator(func):
4      def wrapper():
5          print("Something before the function is called.")
6          func()
7          print("Something after the function is called.")
8      return wrapper
9
10 def say_whee():
11     print("Whee!")
12
13 say_whee = my_decorator(say_whee)
14
15 say_whee() # what  does this print?
```

python decorators

```
1  # a decorator is a function that takes takes another function
2  #  as an argument, and returns some variation of that function
3  def my_decorator(func):
4      def wrapper():
5          print("Something before the function is called.")
6          func()
7          print("Something after the function is called.")
8      return wrapper
9
10 @my_decorator
11 def say_whee():
12     print("Whee!")
13
14 # what does this print?
15 say_whee()
```

python decorators

```
1  def do_twice(func):
2      def wrapper():
3          func()
4          func()
5      return wrapper
6
7  @do_twice
8  def say_whee(): print("Whee!")
9
10 @do_twice
11 def greeting(): print("Hello!")
12
13 # what does this print?
14 say_whee()
15 greeting()
```

python decorators

```
1  def do_twice(func):
2      def wrapper():
3          func()
4          func()
5      return wrapper
6
7  @do_twice
8  def greet(name):
9      print(f"Hello {name}")
10
11 # what does this print?
12 greet('Kendra')
13
14 TypeError: wrapper() takes 0 positional arguments but 1 was given
15
```

python decorators

```
1  def do_twice(func):
2      def wrapper(*args,**kwargs):
3          func(*args,**kwargs)
4          func(*args,**kwargs)
5      return wrapper
6
7  @do_twice
8  def greet(name):
9      print(f"Hello {name}")
10
11 # what does this print?
12 greet('Kendra')
13
14
15
```

python decorators

```
1  def do_twice(func):
2      def wrapper(*args,**kwargs):
3           # if you want decorated func to return something,
4           #  add a return statement to the wrapper
5          func(*args,**kwargs)
6          return func(*args,**kwargs)
7      return wrapper
8
9  @do_twice
10 def greet(name):
11     print(f"Hello {name}")
12     return True
13
14 x = greet('Kendra')
15
```

python decorators with args

```python
def repeat(n): # wrap a decorator inside a parameterized function
    def decorator(func):
        def wrapper(*args,**kwargs):
            for i in range(n):
                func(*args,**kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hey {name}")

@repeat(5)
def whee():
    print("Whee!")
```

```python
# what does this print?
greet('JoeJoe')

# what does this print?
whee()
```

Assignment 10 - due Dec 8th

- create a tic-tac-toe game
  - create tttlib.py file with the following:
    - `BOARD_SPACES = '1','2','3','4','5','6','7','8','9'`
    - `WINNERS = ('1','2','3'), ('4','5','6'), …, ('3','5','7')`
    - `displayBoard(xSpaces:set,oSpaces:set): …`
    - `isWinner(spaces:set) -> bool: …`
    - `getValidMoves(takenSpaces:set): …`
    - `playGame(playerXmove:callable,playerOmove:callable): …`
    - `humanMove(mySpaces:set,opponentSpaces:set) -> str: …`
    - `if __name__=='__main__': playGame(humanMove,humanMove)`
  - create tttai.py with the following:
    - `randomBotMove(mySpaces:set,opponentSpaces:set) -> str`
  - create ttt.py file that
    - runs `playGame(…)` with different players as X and O depending on `sys.argv` …

Assignment 10 - details

- `displayBoard(xSpaces:set,oSpaces:set)` function should clear screen, then display the 9 tic-tac-toe spaces with 1,2,3 being in the top row, 4,5,6 in middle row, and 7,8,9 in bottom row
- display X's and O's instead of numbers depending on which items are found in the sets xSpaces and oSpaces; e.g.,

```
displayBoard({'5','6'},{'7'}) #should clear screen, then print:
```

```
  1  |  2  |  3
-----+-----+-----
  4  |  X  |  X
-----+-----+-----
  O  |  8  |  9
```

Assignment 10 - details

- create ttt.py file that runs playGame(...) with different players as X and O depending on sys.argv; i.e.,
  - python ttt.py h r
    - would play human (as X's) vs randomBot (as O's)
  - python ttt.py r h
    - would play randomBot (as X's) vs human (as O's)
  - python ttt.py h h
    - would play human vs human
  - python ttt.py r r
    - would play randomBot vs randomBot

Assignment 10 - details

- make sure your code is well-documented
  - including your name, docstrings, and comments
- make sure your code is robust
  - if human puts in invalid move, it should be ignored, and human player should be asked for a valid move
- Note:
  - `random.choice(l)` will get you a random element from `l` only if `l` is an indexed sequence (e.g., tuple or list)
  - sets aren't index; if you are trying to get a random element from a set, `s`, you'll need to convert `s` to an ordered type (e.g., list or tuple) before you can get a random item from it with the random.choice function

Assignment 10 - Extra Credit 1

- Add a smarter bot in tttai.py:
  - `oneBotMove(mySpaces:set,opponentSpaces:set) -> str`
    - always makes the winning move if there is one
    - otherwise, always blocks opponent's winning move, if the opponent has a winning move
    - otherwise, plays like randomBot
- Add option to play vs oneBot in ttt.py; e.g.,
  - `python ttt.py o h`
    - would play oneBot v human
  - `python ttt.py r o`
    - would play randomBot v oneBot
  - etc.

Assignment 10 - Extra Credit 2

- Add tttsim.py file
  - tttsim.py will run 1000 tic-tac-toe games between two bots
  - you call it like you would call ttt.py, e.g.,
    - tttsim.py o r
      - runs 1000 oneBot vs randomBot simulations
    - tttsim.py o o
      - runs 1000 oneBot vs oneBot simulations
  - it will print out average numbers of wins, draws, and losses
  - if sys.argv[1] and sys.argv[2] are different (simulation is requested between two different bot types),
    - tttsim.py will display results for both, when first bot is X and second is O, and vice versa

Assignment 10 - Extra Credit 3

- Add an optimal minimax bot in tttbot.py:
  - `minimaxBotMove(mySpaces:set,opponentSpaces:set) -> str`
    - always takes the optimal move, assuming opponent is also optimal
    - never loses
    - https://www.google.com/search?q=minimax+pseudocode