CIS 41B
Advanced Python Programming

Callables

De Anza College
Instructor: Clare Nguyen

Review of Callables

- Any object that can be called, which causes it to execute, is considered a <u>callable</u> in Python.

- Common callables are functions and methods, which are class bound functions.

- Functions typically have a name, and all function names within a module (or all methods in a class) must be unique.

- By using lambda expressions, we can also create short anonymous functions that have no name.

- When we call a function, the list of data values we pass to the function is called the <u>argument list</u>, and each data in the list is an <u>input argument</u>.

  - In Python everything is an object, so every data that is passed to a function is a reference.

  - The big difference is whether the reference is for data that's <u>mutable</u> or <u>immutable</u>.

- When the data are received by the called function, they are stored in appropriate variables in the <u>parameter list</u>.

Function Calls

- In global space, Python must see the function definition first before the function can be called. This is the reason why the call to main (or the first function to run in an application) is at the end of the source file, below the function definition for main.

- But if a functionA is called from inside another functionB (not in global space), then Python doesn't need to see the definition for functionA before the call.

- It is not possible to call a function without entering all required arguments, but it is possible to omit the default arguments.

- When passing arguments to a function, the argument values don't have to be passed in the same order as the parameter list if we name the arguments. When we name an argument in a function call, it is called a keyword argument.

- In an argument list the keyword arguments must come after the positional arguments (the ones without names).

3

Function Definition (1)

- The input to the function is through a comma separated parameter list.
- Input parameters can be required or have a default value.
- Default parameters are listed after all the required parameters in the parameter list.
- Default parameters for *immutable* data types can have any default value that's appropriate for the application.
- Default parameters for *mutable* data types should always default to None. Then inside the body of the function, an appropriate default value can assigned.
- Example:

```
def printStudentInfo(id, name, gpa = 0.0, classList = None) :
    if not classList : classList = [ ]
    # function body continues here...
```

- – The data for gpa is immutable and can default to 0.0 or any float.
- – The data for classList is mutable and can only default to None. Inside the body of the function, classList can be set to [ ] if the caller doesn't pass any input value.
- – classList should not default to [ ] in the parameter list:

```
def printStudentInfo(id, name, gpa = 0.0, classList = [ ])    # Problem!!
```

Function Definition (2)

- Continuing from the example in the previous slide:

  ```
  def printStudentInfo(id, name, gpa = 0.0, classList = None) :
      if not classList : classList = [ ]
      # function continues here...
  ```

  – classList should not be defaulted to [ ] in the parameter list because default values are initialized once, when the function is defined, and not initialized every time the function is called.

  – If classlist was incorrectly defaulted to [ ] in the parameter list:

    • The first time printStudentInfo is called for student1 with no classList input value, classList will default to an empty list. Then let's say student1 classList gets appended with 3 class names.

    • The next time printStudentInfo is called for student2 with no classList input value, then classList will default to the same classList of the first call (now with 3 class names), and student2 ends up with the classList of student1.

  – By initializing classList to [ ] inside the body of the function, each call to printStudentInfo gets its own default empty list.

Variable Length Argument List

- If a function has 2 required parameters and 3 default parameters, then it means the caller can pass a minimum of 2 input arguments and a maximum of 5 input arguments.

- Taking advantage of the packing operator, we can write a function that accepts any number of input arguments. The function is said to accept a variable length argument list.

- Example function header: `def  aFunction(*args, **kwargs) :`

- Example function call: `aFunction(5, 8, option1 = 3, option2 = 6)`

- The variable args (for arguments) is a tuple and the * operator packs all positional arguments into the args tuple.

- The variable kwargs (for keyword arguments) is a dictionary and the ** operator packs all the keyword arguments into the kwargs dictionary with the parameter name as key, argument value as the corresponding value.

- In the example above, args = (5, 8) and kwargs = {option1 : 3, option2 : 6}

- args and kwargs are not the required names, but they are standard names for a variable argument list.

  `def  aFunction(*fred, **wilma)`   would also work, but might not be as readable.

*args, **kwargs Example

- Example of a product calculating function with a variable number of positional arguments:

```
# Function definition:
def product(*args) :
    result = 1
    for arg in args :
        result *= arg
    return result
```

```
# Function call:

product(2, 8)                # return 16
product(1, 2, 3, 4, 5)       # return 120
product(3)                   # return 3
```

- Example of a print function with a variable number of keyword arguments:

```
# Function definition:
def printStudent(name, **kwargs) :
    print("Name:", name)
    for k, v in kwargs.items() :
        print(k, v)
```

```
# Function call:
printStudent("Lucy", major="psychology")
printStudent("Linus", year=2, gpa=3.25)
```

```
# Output:

Name: Lucy
major psychology
Name: Linus
year 2
gpa 3.25
```

Argument Unpacking

- The unpacking operator * can be used for argument unpacking, when the function we call requires multiple input values and we have an iterable with those values.

- Example:
  Given a list:

  ```
  L = (1, 2, 3)
  ```

  and a function definition:

  ```
  def simpleFunction(x, y, z) :
      print(x, y, z)
  ```

- To call simpleFunction and pass myList as an argument:

  ```
  simpleFunction(L)          # Error!   simpleFunction requires 3 arguments
                             # and only one is given
  simpleFunction(*L)         # OK.  L is unpacked to 3 arguments, which match
                             # the 3 parameters
  simpleFunction(8, *myList[1:])    # OK. 8 is the first argument, 2 and 3 are the
                                    # second and third arguments
  ```

Data Type Hints

- Even though Python supports duck typing, for documentation purpose in a large project, it can be helpful to give hints to the type of data that a variable should have.

- This is especially true for function headers in a large project, because the caller of the function most likely isn't the person who coded the function.

- Example: we run into the following function header:

  ```
  def  processData (users, sales, active) :
  ```

  It's hard to tell if users or sales is a list or dictionary or a number, and active might be a boolean or might not be.

- Format for data type hints in a function header:

  ```
  def  functionName( param1 : type, param2 : type) -> return_type  :
  ```

- Example:

  ```
  def  processData(users : list, name : dict, active : bool) -> None  :
  ```

  Now we have a better idea what to pass to the function.

- Note: these are just *hints* about the data type, they are not requirements and Python doesn't enforce the data type in the hints.

Functions as First Class Objects

- Python considers a function to be just like any other data object, such as a list object, an int object, a BankAccount object…

- This means a function can be:
    1. Created and deleted from memory
    2. Referenced as a variable
    3. Passed to another function as input argument, and returned from another function

- In other words Python treats functions as first class objects.
    – This provides a high degree of flexibility when we work with functions.

- In the next slides we will see how this flexibility can make our code cleaner and allow for more language features.

Referencing Function

- When we want to refer to a data object, we use its name:

```
num = 5                   # num is a reference to the memory containing 5
var = num                 # var refers to the same memory as num
print(var)                # output: 5
myList.append(num)        # store the reference num in a list
print(myList[-1])         # output: 5
```

- Because a function is a first order object just like data, we can also refer to a function just like we refer to data:

```
def f() :                 # f is a reference to the memory containing code
    print("a function")

f()                       # running the code at the memory location named f
myfunction = f            # myfunction refers to the same memory as f
myfunction()              # running the same code as above, since the memory
                          # location named myfunction is also named f
myList.append(f)          # store the reference f in a list
myList[-1]()              # running the same code as above, because myList[-1]
                          # is the reference f
```

Referencing Function: Application

- Suppose we have several functions that process a user choice.
  Based on the user choice (1, 2, 3…), a corresponding function is called.

- The standard way to choose a
  function based on the user choice
  is to use an if elif statement:

```
if userInput == 1 :
    doTask1(inputArg)
elif userInput == 2 :
    doTask2(inputArg)
elif userInput == 3 :
    doTask3(inputArg)
else :
    do Task4(inputArg)
```

- But Python lets us shorten the code:

  These are functions

```
taskList = [0, doTask1, dotask2, doTask3, doTask4]
taskList[userInput](inputArg)
```

- The resulting code is shorter and easily extendable.
  If we add a dotask5 function, we simply add it to the taskList and we're
  done. There's no need for another elif clause as in the first solution.

Function as Input Argument

- We have seen that a function can be an input argument with lambda functions. Here's the example we've seen:

```
origList = [1, 2, 3, 4]

def add1(n) :
    return n+1

newList = list(map(add1, origList))            # using a named function
newList = list(map(lambda n: n+1, origList))    # using a lambda expression
```

- Whether the function is a lambda function or a named function, it is an input argument for map.

- Note that when a named function is passed as input argument, only the function name is passed, the argument list is not passed.

```
newList = list(map(add1, origList))        # correct way
newList = list(map(add1(n), origList))     # Error!
```

- The function name is a reference to the memory block that contains the code to the function. This is similar to how a variable name is a reference to the memory block that contains the data object

Function as Input Argument: Application (1)

- Suppose we want to write a function that will generate a list of numbers, where the numbers are doubles of the values 1-5:

```
def generateNums() :
    resultList = [2 * n  for n in range(1,6)]
    print(resultList)
```

- Suppose we also want to generate a list of numbers that are an increment of 10 more than the values 1-5. This means we need to write a similar function, but replace 2*n with 10+n.

```
def generateNums() :
    resultList = [10 + n  for n in range(1,6)]
    print(resultList)
```

- Extending the concept further, what if we want generateNums to work with any math function f(n) to generate the list of numbers? Do we need to write more versions of generateNums?
  - The answer is no. Since functions are first class objects, we only need to write one generateNums function. Then we pass f(n) as an input argument to generateNums and let it call f(n) for us.

Function as Input Argument: Application (2)

- We can write any function f(n).
  Here are two examples:

```
def doubling(n) :
    return 2 * n

def add10(n) :
    return 10 + n
```

- And then we re-write generateNums as:

```
def generateNums(f) :        # argument is a function
    resultList = [f(n) for n in range(1,6)]
    print(resultList)
```

- We can call generateNums and pass to it any f(n):

```
generateNums(doubling)    # output: [2, 4, 6, 8, 10]
generateNums(add10)       # output: [11, 12, 13, 14, 15]
```

Advanced Use of Functions

- Because functions are first class objects, we can create closures, which are the building blocks for decorators.

- Decorators allow us to "decorate" functions, which means to add some new functionality to any function we want.

- There are many uses for decorators in Python. The next slides cover these advanced uses of functions.

Nested Functions

- A nested functionB is a function that is enclosed in another functionA:

```
def  functionA( aStr ) :        # outer functionA definition
                                # in the body of functionA are:
    def functionB() :           #  - nested functionB definition
        print(aStr)
    functionB()                 #  - call of functionB
```

- functionB can access the variable aStr because functionB is enclosed inside the body of functionA, where aStr is in scope.

- When we call functionA and passes in a string:

```
functionA("hello")        # output:  hello
```

  - The input argument "hello" is stored in aStr
  - functionB is called and when it runs, it prints the data in aStr
- The concept of nested function is used in a special way in a closure, covered in the next slides.

Closure

- A closure is a special way to nest functions, with 2 requirements:
    1. The nested function must use an initial value that is set in the outer function
    2. The outer function must return the nested function
- Using the same print "hello" example in the previous slide, but written as a closure:

```
def  functionA( aStr ) :        # outer functionA definition
    def functionB( ) :              # Requirement 1: inner functionB uses
        print(aStr)                 # aStr, which is set in function A

    return functionB               # Requirement 2: return the nested function
```

- Using the closure:

```
f = functionA("hello")       #  f = the return value of functionA = functionB
f()                          # call f(), which means call functionB, output:  hello
```

- When the outer function runs, it receives some initial value, such as aStr. When the nested function runs, it uses the initial value that was set up by the outer function.

Closure Example

- Suppose we have this closure:

```
def  addFrom(start_value) :
     def  adding(n) :              # adding is a closure
          return start_value + n
     return adding
```

- To work with the closure:

```
adder = addFrom(10)        # adder = adding function with start_value of 10
print(adder(2))            # adder (or adding) runs and returns 12, which is printed
print(adder(15))           # adder (or adding) runs and returns 25, which is printed

adder = addFrom(1)         # adder = adding function with start_value of 1
print(adder(2))            # adder (or adding) runs to produce 3, which is printed
```

Using a Closure

- A closure is used when a function needs to receive an external value (such as the start_value of 10 in the previous add example) but we can't pass in the external value in the function call.

- An example of such a function is the key input argument of the sorted function.

  - The input argument for key is a function that sorted will call.

  - This function can have only one input argument which is an element in the list.

  - If we want this function to work with an external value that we set, then a closure is needed to pass this external value to the function.

- More examples of such a function is in a decorator, covered next.

Decorator

- A decorator is a function that accepts an existing function as input argument. It adds new functionality to this existing function's behavior ('decorates it'), and returns this function.

- The decorator uses a closure to call the existing function and to add code for new behavior of the existing function.

- Format of decorator function definition:

```
def  decoratorName (existingFunctionName) :
    def wrapper(*args, **kwargs) :      # closure
        # code to add new behavior to existing function
        result = existingFunctionName(*args, **kwargs)     # call existing function
        # code to add new behavior to existing function
        return result
    return wrapper
```

- When the decorator runs:

```
d = decoratorName (existingFunctionName)       # d = wrapper
d(arguments)              # call d, which means call wrapper. When wrapper runs:
                          #  - code for new behavior runs
                          #  - code for existingFunctionName runs with arguments
```

Decorator At Work

```
def f(n) :                              # existing function f
    print("Hello " * n)                 # prints n number of "Hello" strings

def printStars(fct) :                   # start of decorator
    def prettify(*args, **kwargs) :     # inner fct accepts a variable length arg. list
        print("\n" + "*" * 20)          # code to 'decorate' with a line of *'s
        fct(*args, **kwargs)            # call the existing function, passing in arg list
        print("*" * 20 + "\n")          # code to print another line of *'s
    return prettify

f(n=2)              # without decorator, output:    Hello Hello

d = printStars(f)   # run decorator
d(3)                # with decorator and using *args
                    # output:    ********************
                    #             Hello Hello Hello
                    #            ********************
d(n=2)              # with decorator and using **kwargs
                    # output:    ********************
                    #            Hello Hello
                    #            ********************
```

Using a Decorator

- After we've created a decorator such as printStars in the previous example, we can use it with any existing function that prints to screen.

- In fact, Python makes it easy to decorate a function by providing a shortcut: above each function that needs to be decorated, simply add the decorator name, preceded by @:

```
# From the previous slide:
def f(n) :
    print("Hello " * n)


d = printStars(f)   # remove this line
d(3)
```

```
# Short cut:
@printStars       # add this line
def f(n) :
        print("Hello " * n)


f(3)      # call f directly
```

- The @ symbol means we want to use printStars as a decorator and Python will run the decorator function for us, as long as the decorator function definition is placed above the first @ with the decorator name.

- If we want to apply more than one decorator to a function, we can add multiple @decoratorName lines before the function.

```
@printStars
@countArguments
def functionA() :
        # function body
```

Usage of Decorators

- A decorator provides a fast way to add some temporary, extra functionality to existing functions.

- If we want to decorate multiple functions with printStars, we simply add the decorator line above each of the functions, we don't have to code all the extra print statements to each function.

- Later when we no longer need to print the lines of *'s, we can easily remove the decorator line above the functions, we don't have to remove all the print statements in each function.

- So what are some of the temporary extra functionalities that we would want to add to a function? Here are some common situations:

  - Write to a log file when the function is called. Perhaps the log will help us debug or optimize the code.

  - Check the return value of the function and raise an exception when the return value is above or below a threshold.

  - Time how long a function takes to run, or log some system information before and after the function runs.

  - Add a loop to run the function repeatedly.

24

Decorator and Memoization (1)

- Decorators are commonly used for memoization, an important concept in software optimization.

- Memoization is the caching or storing of the results of an expensive operation (expensive in time or memory or both), so that when we need these results again, we don't have to run the expensive operation. Instead we can be quickly fetched these results from storage.

- A simple example of an expensive operation is a function that generates a Fibonacci number. The Fibonacci sequence starts at the values 0 and 1, and every subsequent number is the sum of the previous 2 numbers. The first few values of the sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

- To calculate the (n+1)th Fibonacci number, a recursive function is the simplest:

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

1st number: fib(0) return 0
2nd number: fib(1), return 1
3rd number: fib(2) = fib(1) + fib(0) = 1
4th number: fib(3) = fib(2) + fib(1) = 2
5th number: fib(4) = fib(3) + fib(2) = 3 …

- As seen from the sample calculations, if we know all the results up to fib(4), then it's a simple addition to calculate fib(5). But if we don't know the previous results. then to run fib(5), we need to re-calculate all the previous results.

Decorator and Memoization (2)

- Without caching or storing results of fib(n), calculating the nth Fibonacci number is a time consuming operation when n is large. And it can take a lot of time to calculate: fib_nums = [fib(i) for i in range(40)]
- The solution is to cache the results of the calculation when we calculate it the first time, then we don't have to keep repeating the same calculation.
- Write the memoize decorator:

```
def memoize(f):      # decorator function
    memo = {}         # use a dictionary as cache
    def helper(x):
        if x not in memo:
            memo[x] = f(x)    # store result the first time we calculate it
        return memo[x]      # return what is stored
    return helper
```

- Use the decorator:
```
@memoize
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- And the great thing about decorators is: we can use this same memoize decorator for any function that can benefit from caching results.

Decorator and Abstract Base Class

- In addition to user-defined decorators, Python also has built-in decorators.
- A common Python decorator used in OOP is @abstractmethod
  - An abstract class is a superclass in an inheritance hierarchy. The abstract class sets a common public interface for all its subclasses by defining abstract methods that serve as the public interface. These abstract methods require all subclasses to implement the interface in the same way, thus giving all the subclasses the same 'look and feel' to the user.
  - The abstract methods in the superclass turn it into an <u>a</u>bstract <u>b</u>ase <u>c</u>lass or an ABC.
- To write an abstract base class:

```
from abc import ABC, abstractmethod

class  MyAbstrClass(ABC):        # our abstract class is derived from abc's class
     #  constructor and other methods go here

    @abstractmethod                          # this decorator turns the do_something
    def do_something(self):              # method into a pure abstract method,
        raise NotImplementedError    # which turns MyAbstrClass into an ABC
```

Decorator and Class Property

- Another common Python decorator used in OOP is @property
  - A @property decorator works with a getter method in a class.
  - A getter method returns an instance attribute of the class, since the user of the class should not access the instance attribute directly.
  - By using the decorator, the instance attribute can appear to be accessible to the user of the class.

```
class  Person:        # our abstract class is derived from abc's class
      #  constructor and other methods go here

    @property                          # this decorator turns the getter method name()
    def name (self):                   # into a property of the class, and the user can
        return self._name        # access the _name attribute more conveniently


p = Person("Guido")        # create Person with "Guido" as the name attribute
print(p.name)               # "Guido" is printed
```

- List of decorators that have been submitted to python.org and are useful enough to make it to the list.

Up Next: numpy