COMP 1023 Introduction to Python Programming

Functions (Part II)

Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Scope of Variables

- The scope of a variable is the part of the program where the variable can be referenced.
- A variable created inside a function is referred to as a local variable.
  - Local variables can only be accessed within the function.
  - The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.
- A variable created outside all functions is referred to as a global variable.
  - Global variables are accessible to all functions in their scope.

```
1  x = 1  # Global variable
2  def f():
3      y = 2  # Local variable
4      print(x)
5      print(y)
6
7  f()
8  print(x)
9  # y is out of scope
10 print(y)
```

Output:

```
1
2
1
Traceback (most recent call last):
  File "Example.py", line 10, in <module>
    print(y)
NameError: name 'y' is not defined
```

Scope of Variables

```
1  x = int(input("Enter an integer: "))
2  if x > 0:
3      y = 4
4  # An error if y is not created
5  print(y)
```

Output:

```
Enter an integer: 10
4
```

Output:

```
Enter an integer: -5
---------------------------------------------
NameError     Traceback (most recent call last)
      2 if x > 0:
      3     y = 4
      4 # An error if y is not created
----> 5 print(y)

NameError: name 'y' is not defined
```

- The variable y is created if $x > 0$ in the example.
- If the user enters a positive value for x (line 1), the program runs fine.
- But if the user enters a non-positive value, line 4 produces an error because y is not created.

Scope of Variables

- We can bind a local variable to the global scope.
- We can also create a variable inside a function and use it outside the function.
- To do so, use the `global` statement.

```
1  # Filename: global_example1.py
2  def create_global():
3      global x
4      x = 7
5      print(x)  # Print 7
6
7  create_global()
8  print(x)  # Print 7
9
```

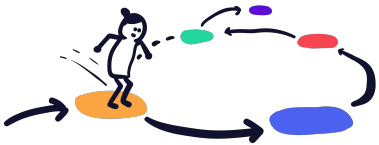
- A global variable x is created in line 3.
- The x in line 8 refers to the global variable x created in line 3.

```
1  # Filename: global_example2.py
2  x = 1
3  def increase():
4      global x
5      x += 1
6      print(x)  # Print 2
7
8  increase()
9  print(x)  # Print 2
```

- A global variable x is created in line 2.
- The x in line 4 is bound to the global variable, meaning that x in the function is the same as x outside the function.

Scope of Variables

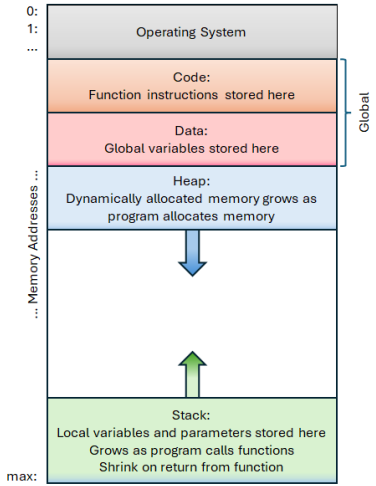
```
# Filename: global_example3.py
x = 20  # x is a global variable

def my_function():
    print(x)    # Accessing the global variable

def modify_global():
    global x    # Commenting out this line
                # results in an error
    x = x + 30  # Modifying the global variable

my_function()   # Print 20
modify_global()
print(x)        # Print 50
```



If you comment out the line `global x`, the statement `x = x + 30` inside `modify_global` is treated as an attempt to assign a value to a local variable `x`. Since `x` has not been defined as a local variable within `modify_global`, Python raises an UnboundLocalError when it tries to evaluate `x + 30` because it cannot find a local variable `x` that has been assigned a value yet.

Memory

- Stack frame
  - A block of memory that stores information about a function call.
  - When a function is called, a new stack frame is created to hold its local variables, parameters, and return address.
  - When the function finishes executing, its stack frame is removed from the stack.
- Global frame
  - The top-level scope in a Python program.
  - It contains global variables and functions that can be accessed from anywhere in the program.
  - When the Python interpreter starts, it creates a global frame for the entire program.
- Heap
  - A region of memory used for dynamic memory allocation.
  - In Python, this is where objects (like integers, floats, lists, dictionaries, and user-defined classes) are stored when they are created.
  - Memory in the heap is managed automatically by Python, and objects can have a longer lifetime than the functions that created them.



Name Conflicts

```
1  # Filename: name_conflict.py
2
3  x = 1          # Global variable
4
5  def f():
6      x = 2      # Local variable
7      print(x)   # Print 2
8
9  f()
10 print(x)       # Print 1
```

- A global variable x is created in line 3.
- A local variable with the same name, x, is created in line 6. From this point on, the global variable x is not accessible within the function.
- Outside the function, the global variable x is still accessible. So, in line 10, it prints 1.

Modularizing Code

- In Python, we can place the function definition into a file called module with the file-name extension .py.
- The module can be later imported into a program for reuse.
- The module file should be placed in the same directory as your other programs.
- A module can contain more than one function.

Question

What happens if we define two functions with the same name in a module?
There is no syntax error in this case, but the latter function definition prevails.

Modularizing Code

```python
# Filename: gcd_function.py

# Return the gcd of two integers
def gcd(n1, n2):
    result = 1  # Initial gcd is 1
    k = 2       # Possible gcd is 2
    while k <= n1 and k <= n2:
        if n1 % k == 0 and n2 % k == 0:
            result = k  # Update gcd
        k += 1
    return result  # Return gcd
```

Output:

```
First integer: 45
Second integer: 75
The GCD for 45 and 75 is 15
```

```python
# Filename: test_gcd_function1.py

# Import the module
from gcd_function import gcd

# Prompt the user to enter two integers
n1 = int(input("First integer: "))
n2 = int(input("Second integer: "))
print("The GCD for", n1, "and", n2,
      "is", gcd(n1, n2))
```

```python
# Filename: test_gcd_function2.py

# Import the module
import gcd_function

# Prompt the user to enter two integers
n1 = int(input("First integer: "))
n2 = int(input("Second integer: "))
print("The GCD for", n1, "and", n2,
      "is", gcd_function.gcd(n1, n2))
```

Modularizing Code

```python
# Filename: prime_number_function.py
# Check whether a number is prime
def is_prime(number):
    divisor = 2
    while divisor <= number / 2:
        # If true, number is not prime
        if number % divisor == 0:
            return False
        divisor += 1
    return True  # number is prime
```

```python
# Filename: prime_number_function.py (Continued)
def print_prime_numbers(number_of_primes):
    NO_OF_PRIMES_PER_LINE = 10  # 10 per line
    count = 0  # Count the number of prime numbers
    number = 2  # Number to be tested for primeness

    # Repeatedly find prime numbers
    while count < number_of_primes:
        if is_prime(number):
            count += 1  # Increase the count
            print(number, end=" ")
            if count % NO_OF_PRIMES_PER_LINE == 0:
                print()  # Move to the next line
        number += 1
```

```python
# Filename: test_print_prime_numbers.py

# Import the module
from prime_number_function import print_prime_numbers

print("The first 50 prime numbers are:")
print_prime_numbers(50)
```

Passing Lists to Functions

- When passing a list to a function, the contents of the list may change after the function call since a list is a mutable object.

```
# Filename: passing_list_to_func1.py

def m(number, numbers):
    number = 1001       # Assign a new value to number
    numbers[0] = 5555   # Assign a new value to numbers[0]

def main():
    x = 1
    y = [1, 2, 3]
    m(x, y)
    print("x is", x)          # Print x is 1
    print("y[0] is", y[0])    # Print y[0] is 5555

if __name__ == "__main__":
    main()
```

Passing Lists to Functions

```
# Filename: passing_list_to_func2.py
def add(x, l = []):  # l is empty by default
    if x not in l:
        l.append(x)
    return l

def main():
    list1 = add(1)
    print(list1)   # Print [1]
    list2 = add(2)
    print(list2)   # Print [1, 2]
    list3 = add(3, [11, 12, 13, 14])
    print(list3)   # Print [11, 12, 13, 14, 3]
    list4 = add(4)
    print(list4)   # Print [1, 2, 4]

if __name__ == "__main__":
    main()
```

- add(1) is invoked. The function appends 1 to list l if x is not in the list. When the function is executed for the first time, the default value [] for the argument l is created. This default value is created only once. We append 1 to l, i.e., [], so l becomes [1].
- add(2) is invoked. l is now [1], not [], because l is created only once. After add(2) is executed, l becomes [1, 2].
- add(3, [11, 12, 13, 14]) is invoked. The list [11, 12, 13, 14] is passed to l. After appending 3 to l, l becomes [11, 12, 13, 14, 3].
- add(4) is invoked. Since the default list now is [1, 2], after invoking add(4), the default list becomes [1, 2, 4].

In Python, when you define a function with a default parameter that is a mutable data type (e.g., list), that default parameter is created once when the function is defined, not each time the function is called. This means that if you modify the list during one function call, the changes will persist in subsequent calls to the function.

Avoiding Troubles

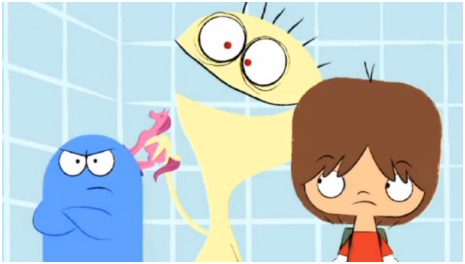
```python
# Filename: passing_list_to_func2_fix.py
def add(x, l=None):  # Set default value to None
    if l is None:
        l = []
    if x not in l:
        l.append(x)
    return l

def main():
    list1 = add(1)
    print(list1)   # Print [1]
    list2 = add(2)
    print(list2)   # Print [2]
    list3 = add(3, [11, 12, 13, 14])
    print(list3)   # Print [11, 12, 13, 14, 3]
    list4 = add(4)
    print(list4)   # Print [4]

if __name__ == "__main__":
    main()
```

In this revised example, each call to add will create a new list, preventing any unintended persistence of changes.

Returning a List from a Function

- When a function returns a list, the list's reference is returned.

```
# Filename: reverse_list.py
def reverse(l):
    result = [0] * len(l)
    for i in range(0, len(l)):
        result[i] = l[len(l) - 1 - i]
    return result

def main():
    list1 = [1, 2, 3, 4, 5, 6]
    list2 = reverse(list1)
    print(list2)  # Print the reversed list

if __name__ == "__main__":
    main()
```



Output:

```
[6, 5, 4, 3, 2, 1]
```

Passing Any Number of Parameters to Functions in Python

- Python allows functions to accept any number of parameters.
- This flexibility enables functions to handle a varying amount of input data.
- Using `*args` and `**kwargs`:
  - `*args` allows passing a variable number of non-keyword arguments.
  - `**kwargs` allows passing a variable number of keyword arguments.

```python
# Filename: example_any_num_parameters.py
def example_function(*args, **kwargs):
    print("Arguments:", args, type(args))
    print("Keyword Arguments:", kwargs, type(kwargs))

def main():
    example_function(1, 2, 3, name='Alice', age=30)

if __name__ == "__main__":
    main()
```

Output:

```
Arguments: (1, 2, 3) <class 'tuple'>
Keyword Arguments: {'name': 'Alice', 'age': 30} <class 'dict'>
```

Passing Two-Dimensional Lists To Functions

```python
# Filename: matrix_accumulate.py
def get_matrix():
    matrix = []  # Create an empty list
    number_of_rows = int(input("Enter the number of rows: "))
    for row in range(number_of_rows):
        s = input("Enter row " + str(row) + ": ")
        matrix.append([float(x) for x in s.split()])
    return matrix

def accumulate(m):
    total = 0
    for row in m:
        total += sum(row)  # Get the total in the row
    return total

def main():
    m = get_matrix()  # Get a list
    print(m)
    print("\nSum of all elements is", accumulate(m))

if __name__ == "__main__": main()
```

Output:

```
Enter the number of rows: 2
Enter row 0: 2 3
Enter row 1: 4 5
[[2.0, 3.0], [4.0, 5.0]]

Sum of all elements is 14.0
```

Lambda Functions

- Lambda functions are anonymous expressions, meaning they have no name unless explicitly assigned to a variable.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda <arguments> : <expression>
```

- Example:

```
(lambda x, y: x + y)(1, 2)  # Return 3

func = lambda x, y: x + y
print(func(1, 2))  # Print 3
```

Note

A lambda function can take no parameters. The following example illustrates this:

```
# Lambda function that returns a greeting message
greet = lambda: "Hello, welcome to COMP 1023!"
print(greet())  # Output: Hello, welcome to COMP 1023!
```

Common Use Cases for Lambda Functions: filter(), map(), and reduce()

- Lambda functions are often used in functional programming, particularly with functions like filter(), map(), and reduce(), which take other functions as arguments to process elements in a collection.
- filter():

  `filter(<function>, <iterable>)`

  It applies the given function to each item of the iterable. If the returned value is True, the item is kept in a new iterable; otherwise, it is excluded.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # Print [2, 4, 6, 8]
```

Common Use Cases for Lambda Functions: filter(), map(), and reduce()

- map():

  `map(<function>, <iterable>)`

  It applies the given function to each item of the iterable and keeps the returned value in a new iterable.

```
fruits = ['apple', 'banana', 'cherry']
lengths = list(map(lambda x: len(x), fruits))
print(lengths)       # Print [5, 6, 6]
```

- reduce():

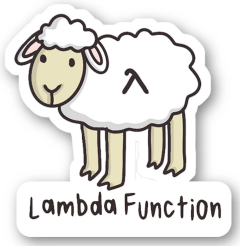
  `reduce(<function>, <iterable>)`

  It applies the given function to the first two items of the iterable, then takes the returned value and applies the function to it and the next item, and so on, until all items are processed.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
result = reduce(lambda x, y: x + y, numbers)
print(result)        # Print 36
```

Lambda Functions

- Lambda functions are not inherently faster than standard functions, as both are compiled to similar bytecode.
- However, they can slightly reduce overhead in cases where defining a full function would add unnecessary boilerplate.
- Lambda functions can be used inline as anonymous functions when passed directly to higher-order functions like map() or filter().
- This avoids the need to define and reference a separate named function, reducing both boilerplate code and lookup overhead.



Key Terms

- argument
- caller
- default argument
- function
- function header
- global variable
- immutable objects
- keyword arguments
- local variable
- None
- None function
- parameter
- positional arguments
- return value
- void function

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. A variable created in a function is called a _____. Its scope starts from its _____ and exists until the function _____.
2. _____ are created outside all functions and are accessible to all functions in their scope.

Answer: 1. local variable; creation; returns, 2. Global variables

Further Reading

- Read Chapter 6 of "Introduction to Python Programming and Data Structures" textbook.

That's all!

Any questions?