COP 3035

Intro Programming in Python

Summer 2024

Lecture 14 – part 1

Homework 4 – 06/28/24
Lab 7 – 07/01/24
Homework 5 07/05/24

Lecture 14 – part 2

Review

Review

Functions

Functions exercises

What is a function ?

- A function is a valuable tool that groups a set of statements together, allowing them to be executed multiple times.
- This prevents us from having to write the same code repeatedly.

Function Syntax

```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes.
    When you call help() on your function it will be printed out.
    '''
    # Do stuff here
    # Return desired result
```

Lecture 14 – part 3

Lambda Expressions

Lambda Expressions

- Lambda expressions allow us to create "anonymous" functions.
- We can quickly make ad-hoc functions without needing to properly define a function using def.
- Lambda's body is a single expression, not a block of statements.

```
def square(num):
    result = num**2
    return result
```

```
lambda num: num ** 2
```

```
def square(num): return num**2
```

```
square = lambda num: num **2
```

Lecture 14 – part 3

default values, *args, **kwargs

The print function

Same as *args

Default values

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

matplotlib.pyplot.plot

matplotlib.pyplot.plot(**args, scalex=True, scaley=True, data=None, **kwargs*)

[source]

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

Default Values in Function Parameters

- Functions can have **default values for parameters**.
- These defaults are used if no argument is passed for that parameter.

<u>Syntax:</u>

```
def function_name(param=default_value):
```

- Default values make function arguments optional.
- If an argument is passed, it overrides the default value

```python
def greet(name, message="Hello"):
    return f"{message}, {name}!"

print(greet("Alice"))             # Output: Hello, Alice!

print(greet("Alice", "Goodbye"))  # Output: Goodbye, Alice!
```

Understanding ***args** in Python

*args allows a function to accept any number of **positional arguments**.

<u>Syntax:</u>

```
def function_name(*args):
```

How it works:

- Inside the function, args is accessible as a <u>tuple</u>.
- Enables flexible function calls without specifying the exact number of arguments.

```
def test_function(*args):
        for i,a in enumerate(args):
                print(f'Index: {i}, Argument: {a}’)
return 0
```

```
a = test_function(1,2,3,4,5)
```

```
Index: 0, Argument: 1
Index: 1, Argument: 2
Index: 2, Argument: 3
Index: 3, Argument: 4
Index: 4, Argument: 5
```

```
a = test_function('salary',200,10)
```

```
Index: 0, Argument: salary
Index: 1, Argument: 200
Index: 2, Argument: 10
```

Understanding **kwargs in Python

kwargs allows a function to accept any number of **keyword arguments.

<u>Syntax:</u>

```
def function_name(**kwargs):
```

How it works:

- Inside the function, kwargs is accessible as a dictionary.
- Facilitates receiving named arguments not predefined in function parameters.

```python
def person_details(**kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")


person_details(name="John", age=30, city="New York")
```

```
name: John
age: 30
city: New York
```

Lecture 14 – part 4

Scope

Python Scope with the LEGB Rule

The **LEGB rule** is a well-established guideline in the Python community to comprehend the **order** in which Python searches for variable names. The acronym stands for:

L: Local

- Names assigned within a function (def or lambda).
- Not declared as global within that function.

E: Enclosing function locals

- Names in the local scope of any and all enclosing functions (def or lambda), from innermost to outermost.

G: Global (module)

- Names assigned at the top-level of a module.
- Or declared as global within a def in the file.

B: Built-in (Python)

- Names preassigned in Python like open, range, SyntaxError, etc.

In simple terms:

- By default, name assignments will create or change local names.
- Name references search through, at most, four scopes. These scopes, in order, are:
  - local
  - enclosing functions
  - global
  - built-in
- Names declared in **global** and **nonlocal** statements map the assigned names to the enclosing module and function scopes.