
`Functions`

---

1. The function header always starts with the `def` keyword, which indicates that this is a **function definition**.
2. The header always end with a colon `:`.
3. We can add default arguments in a function to have default values for parameters that are unspecified in a function call.

```
def cylinder_volume(height, radius=5):
    pi = 3.14159
    return height * pi * radius ** 2
```

note: It is possible to pass values in two ways - **by position** and **by name**. Each of these function calls are evaluated the same way.

```
cylinder_volume(10, 7)  # pass in arguments by position
cylinder_volume(height=10, radius=7)  # pass in arguments by name
```

```
It is best to define variables in the smallest scope they will be needed in.
```

Documentation

```
def population_density(population, land_area):
    """Calculate the population density of an area.

        INPUT:
    population: int. The population of that area
    land_area: int or float. This function is unit-agnostic, if you pass in values in terms
    of square km or square miles the function will return a density in those units.

    OUTPUT:
```

```
    population_density: population / land_area. The population density of a
particular area.
    """

    return population / land_area
```

`Lambda`

```
def multiply(x, y):
    return x * y
```

can be reduced to:

```
multiply = lambda x, y: x * y
```

Components of a Lambda Function

1. The `lambda` keyword is used to indicate that this is a lambda expression.
2. Following `lambda` are one or more arguments for the anonymous function separated by commas, followed by a colon `:`. Similar to functions, the way the arguments are named in a lambda expression is arbitrary.
3. Last is an expression that is evaluated and returned in this function. This is a lot like an expression you might see as a return statement in a function.

note: With this structure, lambda expressions aren’t ideal for complex functions, but can be very useful for short, simple functions.

`want a function with many returns say no more :D here comes the GENERATORS`

```

 def simple_generator_function():
yield 1
yield 2
yield 3
```

And here are two simple ways to use it:

```
>>> for value in simple_generator_function():
>>>     print(value)
1
2
3
>>> our_generator = simple_generator_function()
>>> next(our_generator)
1
>>> next(our_generator)
2
>>> next(our_generator)
3
```

- `generators` are used to *generate* a series of values
- `yield` is like the `return` of `generator functions`
- The only other thing `yield` does is save the "state" of a `generator function`
- A `generator` is just a special type of `iterator`
- Like `iterators`, we can get the next value from a `generator` using `next()`
  - `for` gets values by calling `next()` implicitly

`Exceptions`

Exceptions occur when unexpected things happen during execution of a program, even if the code is syntactically correct. There are different types of built-in exceptions in Python, and you can see which exception is thrown in the error message.

```
try:
    # some code
except (ValueError, KeyboardInterrupt):
    # some code
```

Or, if we want to execute different blocks of code depending on the exception, you can have multiple `except` blocks.

```
try:
    # some code
except ValueError:
    # some code
except KeyboardInterrupt:
    # some code
```

- `finally`: Before Python leaves this `try` statement, it will run the code in this `finally` block under any conditions, even if it's ending the program. E.g., if Python ran into an error while running code in the `except` or `else` block, this `finally` block will still be executed before stopping the program.

`File Operations`

Reading a File

```
f = open('my_path/my_file.txt', 'r')
file_data = f.read()
f.close()
```

Writing to a File

```
f = open('my_path/my_file.txt', 'w')
f.write("Hello there!")
f.close()
```

Appending

```
files.append(open('some_file.txt', 'r'))
```

With

Python provides a special syntax that auto-closes a file for you once you're finished using it.

```
with open('my_path/my_file.txt', 'r') as f:
    file_data = f.read()
```

Importing

To import multiple individual objects from a module:

```
   from module_name import first_object, second_object
```

To rename a module:

```
    import module_name as new_name
```

`requirements.txt` File

Larger Python programs might depend on dozens of third party packages. To make it easier to share these programs, programmers often list a project's dependencies in a file called requirements.txt. This is an example of a requirements.txt file.

```
beautifulsoup4==4.5.1

bs4==0.0.1

pytz==2016.7

requests==2.11.1
```

Using a main block

To avoid running executable statements in a script when it's imported as a module in another script, include these lines in an `if __name__ == "__main__"` block. Or alternatively, include them in a function called main() and call this in the `if main` block.