Functions

Functions

A function is a reusable block of code. Functions

- have names (usually),
- contain a sequence of statements, and
- return values, either explicitly or implicitly.

We've already used several built-in functions. Today we will learn how to define our own.

Hello, Functions!

We define a function using the def keyword:

```
1 def greet():
2     print('Hello')
```

Once the function is defined, you can call it:

```
1 >>> greet()
2 Hello
```

Active Review

- What happens if you evaluate `greet` (without the `()`) in the Python REPL?

Defining Functions

The general form of a function definition is

```
def <function_name>(<parameter_list>):
    <function_body>
```

- The first line is called the header.
- `function_name` is the name you use to call the function.
- `parameter_list` is a list of parameters to the function, which may be empty.
- `function_body` (also called a suite in Python) is a sequence of expressions and statements.

Function Parameters

Provide a list of parameter names inside the parentheses of the function header, which creates local variables in the function.

```
1 def greet(name):
2     g = "Hello, " + name + "!"
3     print(g)
```

Then call the function by passing *arguments* to the function: values that are bound to parameter names.

Here we pass the value `'Dolly'`, which is bound to `greet`'s parameter `name` and printed to the console by the code inside `greet`.

```
1 >>> greet('Dolly')
2 Hello, Dolly!
```

Variable Scope

Parameters are local variables. They are not visible outside the function.

```
def greet(name):
    g = "Hello, " + name + "!"
    print(g)
>>> greet('Dolly')
Hello, Dolly!
>>> name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
```

Global variables are visible outside the function and inside the function.

```
>>> global_hello = 'Bonjour'
>>> global_hello
'Bonjour'
>>> def say_global_hello():
...     print(global_hello)
...
>>> say_global_hello()
Bonjour
```

Shadowing Global Variables

Local variables shadow global variables.

```
>>> x = 1
>>> def f():
...     x = 2
...     print("local x:", x)
...     print("global x:", globals()["x"])
...
>>> f()
local x: 2
global x: 1
```

A function parameter is a local variable.

```
>>> name = 'Meat Loaf'
>>> def mayhem(name):
...     print(f"His name is {name}.")
...
>>> mayhem('Robert Paulson')
His name is Robert Paulson.
>>> name
'Meat Loaf'
```

Scopes and Namespaces

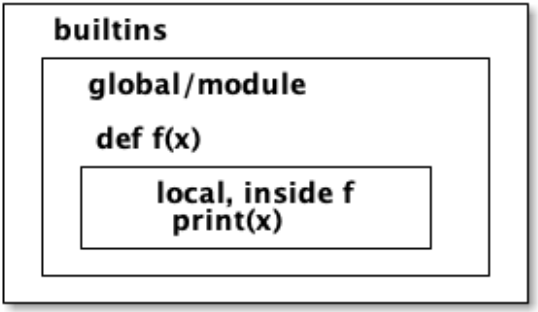
A **namespace** is a mapping from names to values (sometimes also called a *symbol table*).

- Top level, or *global* names (either the Python REPL or a script) are in a namespace called `__main__`.
- Global is also used to refer to names that are defined at the module level, i.e, outside any function or class. They are global to the module.
- Each function *call* also gets a namespace for the local variables in the function.
- These namespaces are hierarchical – name resolution starts with the innermost namespace, which is why local variables "hide" or "shadow" global variables.
- A variable's **scope** is the region of source code in which that variable is visible.
- A namespace contains all the variables in a scope.

Active Review

- Evaluate `globals()["__name__"]` in your Python REPL.

Python Scopes



- Global `x` and the local `x` inside `f` are different.
- `print`, referenced inside `f`, is from the `builtins` namespace.

Python Scope Resolution

Scopes are determined statically but used dynamically. Python determines the value of a variable by searching scopes in the following order (LEGB):

1. Local
2. Enclosing (for nested functions)
3. Global
4. Builtins

Active Review: Python Scope Resolution

Apply the LEGB rule in the following exercises:

- Enter and run the following program. What happens?

```
1  def f():
2      print(x)
3      x = 2
4
5  def g(x):
6      print(x)
7
8  if __name__=='__main__':
9      x = 1
10     f()
11     g(x)
```

- Comment-out the `x = 1` in the `if __name__=='__main__'` block and `x = 2` line in `def f()`. Explain the program's behavior.
- Uncomment the `x = 1` and leave the `x = 2` line in `def f()` commented-out. Explain the program's new behavior.
- Uncomment the `x = 2` and add `global x` as the first line `def f()`. Explain the program's new behavior.

Multiple Parameters

A function can take any number of parameters.

```
>>> def greet(greeting, name):
...     print(greeting + ', ' + name)
...
>>> greet('Greetings', 'Professor Falken')
Greetings, Professor Falken
```

Parameters can be of multiple types.

```
>>> def greet(name, greeting, number):
...     print(greeting * number + ', ' + name)
...
>>> greet('Professor Falken', 'Hello', 2)
HelloHello, Professor Falken
```

Positional and Keyword Arguments

Thus far we've called functions using positional arguments, meaning that argument values are bound to parameters in the order in which they appear in the call.

```
>>> def greet(greeting, name, number):
...     print((greeting + ', ' + name) * 2)
...
>>> greet('Hello', 'Dolly', 2)
Hello, DollyHello, Dolly
```

We can also call functions with keyword arguments in any order.

```
>>> greet(greeting='Hello', number=2, name='Dolly')
Hello, DollyHello, Dolly
```

If you call a function with both positional and keyword arguments, the positional ones must come first.

Default Parameter Values

You can specify default parameter values so that you don't have to provide an argument.

```
>>> def greet(greeting, name='Elmo'):
...     print(greeting + ', ' + name)
...
>>> greet('Hello')
Hello, Elmo
```

If you provide an argument for a parameter with a default value, the parameter takes the argument value passed in the call instead of the default value.

```
>>> greet('Hi', 'Guy')
Hi, Guy
```

Return Values

Functions return values.

```
>>> def double(num):
...     return num * 2
...
>>> double(2)
4
```

If you don't explicitly return a value, `None` is returned implicitly.

```
>>> def g():
...     print("man") # This is not a return!
...
>>> fbi = g()
man # This is a side-effect of calling g(), not a return value
>>> type(fbi)
<class 'NoneType'>
```

Function calls are expressions like any other, that is, a function call has a value, so a function call can appear anywhere a value can appear.

```
>>> double(2) + double(3)
10
```

Variable Argument Lists

You can collect a variable number of positional arguments as a tuple by prepending a parameter name with *

```
>>> def echo(*args):
...     print(args)
...
>>> echo(1, 'fish', 2, 'fish')
(1, 'fish', 2, 'fish')
```

You can collect variable keyword arguments as a dictionary with **

```
>>> def print_dict(**kwargs):
...     print(kwargs)
...
>>> print_dict(a=1, steak='sauce')
{'a': 1, 'steak': 'sauce'}
```

Mixed Argument Lists

And you can do positional and keyword variable arguments together, but the keyword arguments come second.

```
>>> def print_stuff(*args, **kwargs):
...     print(args, kwargs)
...
>>> print_stuff("Pass", "the", a=1, steak='sauce')
{'a': 1, 'steak': 'sauce'}
```

Active Review

- What happens when you evaluate

```
print_stuff("Pass", a=1, steak='sauce', 'the')
```

Inner Functions

If you only need a function inside one other function, you can declare it inside that function to limit its scope to the function where it is used.

```
1  def factorial(n):
2      def fac_iter(n, accum):
3          if n <= 1: return accum
4          return fac_iter(n - 1, n * accum)
5      return fac_iter(n, 1)
6
7  >>> factorial(5)
8  120
```

`fac_iter()` is a (tail) recursive function. Recursion is important for purely functional languages, but a practically-oriented Python-programming engineer will mostly use iteration, higher-order functions and loops, which are more Pythonic. Any recursive computation can be formulated as an imperative computation.

Active Review

- Define the `factorial` function above in your REPL and evaluate the following calls:

```
1  factorial(10)
2  factorial(100)
3  factorial(1000)
4  factorial(10000)
```

Conclusion

- Functions are the primary way we break a program into reusable pieces.
- Use functions liberally.