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 seen functions in our tour of Python. In this lesson we'll dive deeper.

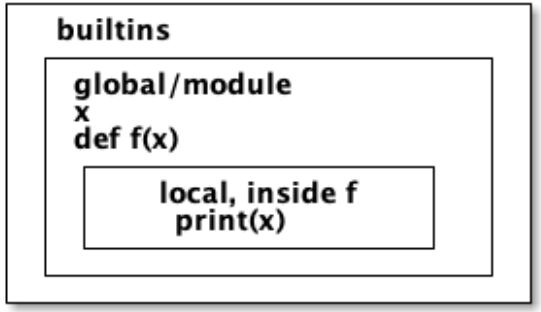
Defining Functions

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.

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

Each scope is a *namespace*, a.k.a. environment or context. Namespaces can be thought of as dictionaries that map (variable) names to values.

Active Review

- Evaluate `globals()["__name__"]` in your Python REPL.
- Import the `math` module.
- Evaluate `globals()["math"]` in your Python REPL.

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.

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, which means that a function call is an expression.

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

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

```
>>> def dubbel(num):
...     print(num * 2)
...
>>> res = dubbel(3)
6
>>> type(res)
<class 'NoneType'>
```

Active Review

- Define the `double` and `dubbel` functions above.
- Evaluate `double(2)+ double(3)`. Explain how it works.
- Evaluate `dubbel(2)+ dubbel(3)`. Explain the result.

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'}
```

Keyword-Only Arguments

If a function has parameters following a varargs, the remaining arguments must be passed as keyword arguments.

Active Review

- Look up the documentation for the built-in `print` function in a Python REPL.
- Execute `print("Hello")` and note the output.
- Execute `print("Hello", "world")` and note the output.
- Execute `print("Hello", "world", end="")` and note the output.
- Execute `print("Hello", "world", end="")`.
  - Why do you get the output you get?
  - How does the documentation for `print` alert you to this fact?

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.
- Python offers very flexible function call semantics.
- Be aware that all functions return values.
  - If no `return` statement, `None` implicitly returned.