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:

```
>>> def greet():
...     print('Hello')
...
```

(blank line tells Python shell you're finished defining the function)

Once the function is defined, you can call it:

```
>>> greet()
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.

```
>>> def greet(name):
        g = "Hello, " + name + "!"
...     print(g)
...
```

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

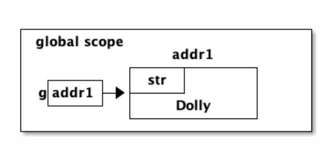
Here we pass the value 'Dolly', which is bound to greet's parameter name and printed to the console by the code inside greet.

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

Function Call Semantics

```
>>> g = "Dolly"
```

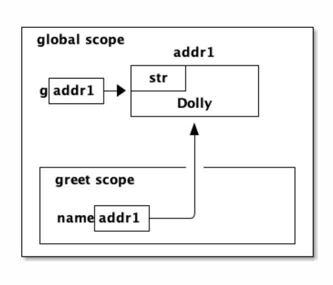
Creates a global value[a].



- Is g a good variable name here?

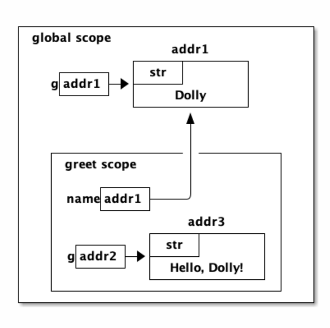
```
>>> greet(g)
```

Passes argument g *by value*, that is, the object pointer in g is copied to `greet`'s `name` parameter.



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

Notice that `greet`'s g shadows the global g.



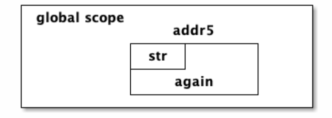
[a] Since `str` is a sequence data structure, this memory image is a slight simplification.

Strict Argument Evaluation

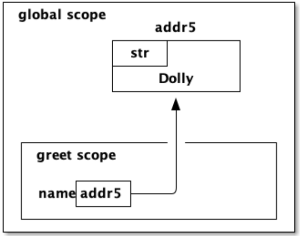
Arguments to functions are evaluated strictly, meaning that they are evaluated before control is transferred to the function body.

```
>>> greet(’again’)
Guten Tag!
```

This creates a temporary `str` object pointing to the `Sequence` value `’again’`

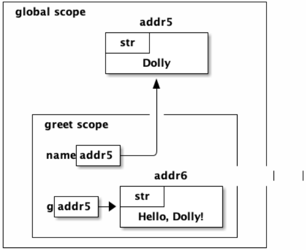


and passes a reference to that object to the function.



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

Then, as before, the local `g` object is created.



Variable Scope

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

```
>>> 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
```

- Tip: evaluate `globals()["__name__"]` in the Python REPL.

A function parameter is a local variable.

```
>>> name = ’Hi ya!’
>>> def greet(name):
...     print(name)
...
>>> name
’Hi ya!’
>>> greet(’Hello’)
Hello
```

Namespaces

Every place where a variable can be defined is called a namespace or a frame (sometimes also called a symbol table, which is how namespaces are implemented by compilers and interpreters).

- Top level, or global names (either the Python REPL or a script) are in a namespace called `__main__`.
- 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.

Redefining Names

A function a kind of variable. If you define a function with the same name as a variable, it re-binds the name, and vice-versa.

```
>>> global_hello = 'Bonjour'
>>> def global_hello():
...     print('This is the global_hello() function.')
...
>>> global_hello
<function global_hello at 0x10063b620>
```

Python Scope Gotchas

Python has notoriously weird scoping rules.

Muliple 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, name, number):
...     print(name * number + ’, ’ + name)
...
>>> greet(’Professor Falken’, ’Greetings’, 2)
GreetingsGreetings, 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(’Professor Falken’, ’Greetings’, 2)
```

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’}
```

And you can do both, but the keword 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

Information hiding is a general principle of software engineering. If you only need a function in one place, inside another function, you can declare it inside that function so that it is visible only in that function.

```
def factorial(n):
   def fac_iter(n, accum):
       if n <= 1:
           return accum
       return fac_iter(n - 1, n * accum)
   return fac_iter(n, 1)

>>> factorial(5)
120
```

`fac_iter()` is a (tail) recursive function. Recursion is important for computer scientists, 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:

```
factorial(10)
factorial(100)
factorial(1000)
factorial(10000)
```

Conclusion

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