# 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')
3  ...
```

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

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

```
1  def <function_name>(<parameter_list>):
2      <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)
4  ...
```

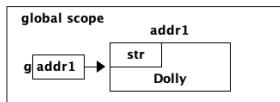Then call the function by passing *arguments* to the function: values that are bound to parameter names.

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!
```

# Function Call Semantics
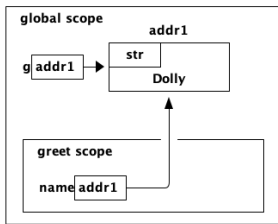
```
>>> g = "Dolly"
```

Creates a global value[a].



▶ Is `g` a good variable name here?

---

[a]Since `str` is a sequence data structure, this memory image is a slight simplification.
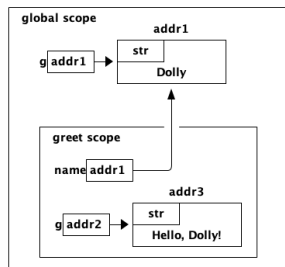
```
>>> greet(g)
```

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



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

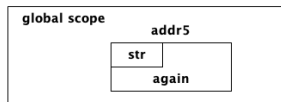Notice that `greet`'s `g` shadows the global `g`.
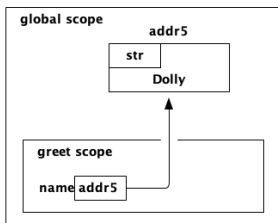
# Strict Argument Evaluation

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

```
1  >>> greet('again')
2  Guten Tag!
```

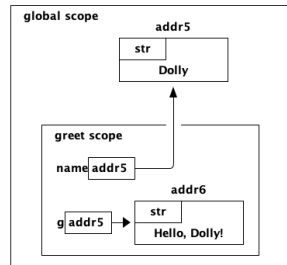This creates a temporary `str` object pointing to the `Sequence` value `'again'`



and passes a reference to that object to the function.



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

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

# Variable Scope

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

```
1  >>> name
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  NameError: name 'name' is not defined
```

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

```
1  >>> global_hello = 'Bonjour'
2  >>> global_hello
3  'Bonjour'
4  >>> def say_global_hello():
5  ...     print(global_hello)
6  ...
7  >>> say_global_hello()
8  Bonjour
```

# Shadowing Global Variables

Local variables shadow global variables.

```
1  >>> x = 1
2  >>> def f():
3  ...     x = 2
4  ...     print("local x:", x)
5  ...     print("global x:", globals()["x"])
6  ...
7  >>> f()
8  local x: 2
9  global x: 1
```

A function parameter is a local variable.

```
1  >>> name = 'Hi ya!'
2  >>> def greet(name):
3  ...     print(name)
4  ...
5  >>> name
6  'Hi ya!'
7  >>> greet('Hello')
8  Hello
```

## Active Review

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

# 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.

```
1  >>> global_hello = 'Bonjour'
2  >>> def global_hello():
3  ...       print('This is the global_hello() function.')
4  ...
5  >>> global_hello
6  <function global_hello at 0x10063b620>
```

# Muliple Parameters

A function can take any number of parameters.

```
1  >>> def greet ( greeting , name ):
2  ...      print ( greeting + ', ' + name )
3  ...
4  >>> greet ( 'Greetings', 'Professor Falken' )
5  Greetings , Professor Falken
```

Parameters can be of multiple types.

```
1  >>> def greet ( name , name , number ):
2  ...      print ( name * number + ', ' + name )
3  ...
4  >>> greet ( 'Professor Falken', 'Greetings', 2)
5  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.

```
1  >>> def greet(greeting, name, number):
2  ...        print((greeting + ', ' + name) * 2)
3  ...
4  >>> greet('Professor Falken', 'Greetings', 2)
```

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

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

```
1  >>> def greet(greeting, name='Elmo'):
2  ...     print(greeting + ', ' + name)
3  ...
4  >>> greet('Hello')
5  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.

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

# Return Values

Functions return values.

```
1  >>> def double ( num ):
2  ...     return num * 2
3  ...
4  >>> double (2)
5  4
```

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

```
1  >>> def g ():
2  ...     print (" man ") # This is not a return !
3  ...
4  >>> fbi = g ()
5  man # This is a side - effect of calling g () , not a return value
6  >>> type ( fbi )
7  < 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.

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

# Variable Argument Lists

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

```
1  >>> def echo(*args):
2  ...     print(args)
3  ...
4  >>> echo(1, 'fish', 2, 'fish')
5  (1, 'fish', 2, 'fish')
```

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

```
1  >>> def print_dict(**kwargs):
2  ...     print(kwargs)
3  ...
4  >>> print_dict(a=1, steak='sauce')
5  {'a': 1, 'steak': 'sauce'}
```

# Mixed Argument Lists

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

```
1 >>> def print_stuff(*args, **kwargs):
2 ...      print(args, kwargs)
3 ...
4 >>> print_stuff("Pass", "the", a=1, steak='sauce')
5 {'a': 1, 'steak': 'sauce'}
```

### Active Review
▶ What happens when you evaluate

```
1 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.

```python
1  def factorial(n):
2      def fac_iter(n, accum):
3          if n <= 1:
4              return accum
5          return fac_iter(n - 1, n * accum)
6      return fac_iter(n, 1)
7
8  >>> factorial(5)
9  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:

```python
1  factorial(10)
2  factorial(100)
3  factorial(1000)
```

# Conclusion

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