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
```

> 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` 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.

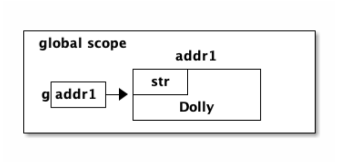
Here we pass the value 'Hello', which is bound to `greet`'s parameter `name` and printed to the console by the code inside `greet`.

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

Function Call Semantics

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

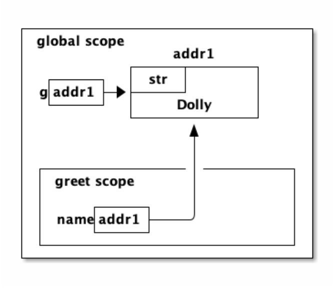
Creates a global value[a].



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

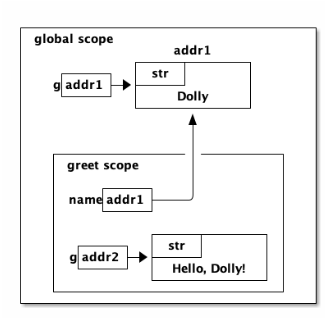
```
>>> greet(g)
```

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



```
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.

Here we pass the value 'Guten Tag!':

```
>>> greet('Guten Tag!')
Guten Tag!
```

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.

Memory Model With Function Calls

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

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

```
>>> greet(name='Hello', number=2, name='Dolly')
HelloHello, 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(name, name='Hello'):
...     print(name + ', ' + name)
...
>>> greet('Elmo')
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('Elmo', 'Hi')
Hi, Elmo
```

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
```

Function Design Recipe

1. Examples
   - What a few representative calls to the function look like in the Python REPL.
     - Think from the function user's perspective.
     - Examples become doctests in the function's docstring.
2. Header
   - Parameter names and types
   - Return type
3. Description
   - Short paragraph (1 or 2 sentences) describing the function's behavior.
4. Body
   - Implement the algorithm (sequence of statements) that accomplishes the function's task, deriving the function's output (return value) and/or effect from the the function's inputs (arguments).
5. Test
   - Test your function on some representative inputs (try to include edge cases).

Writing Function Examples

Let's apply this design recipe in the creation of a simple function to calculate the length of the hypotenuse from the lengths of the two legs (the sides that join in a right angle).

First, decide the name of the function.

- Descriptive word(s)
  - Verbs may imply an imperative function called for its effect, not a return value
    - `print("hello")`, `exit()`
  - Nouns may imply a pure function, a return value derived only from the function's arguments with no side effects
    - `type(1)`, `double(2)`
- Avoid Python keywords or names of library functions.
  - Tip:

```
>>> import keyword
>>> keyword.kwlist # lists all the Python keywords
>>> keyword.iskeyword("foo") # True if "foo" is a keyword
```

- Follow Python's naming conventions.

Hypotenuse Function Examples

We'll name our function `hypotenuse`. General naming tips:

- Only abbreviate if abbreviation is well-known or obvious
  - If you must, form a new abbreviation by eliminating vowels starting from the right, e.g., `format` $\rightarrow$ `formt` $\rightarrow$ `fmt`
- Some abbreviations are idiomatic, e.g., `i` as an loop variable used as an `int` index
- Length of the name should be inversely proportional to its scope
  - Local variables can be short
  - Modules, functions, and classes should have more descriptive names

Our examples:

```
>>> hypotenuse(3, 4)
5
>>> hypotenuse(5, 12)
13
```

Function Headers

The function header includes the function's name and parameter names. We add a type contract, which we document using Python's new (as of 3.5) type hints feature. Here are a few basic types. A full explanation is in PEP 484, including a complete list of types in the `typing` module

::::::::::::: {.columns} ::: {.column width="40%"}

- `int`
- `float`
- `str`

::: ::: {.column width="60%"}

- `List[int]`
- `Tuple[float]`
- `Dict[str, int]`

::: :::::::::::::

Hypotenuse Function Header

Deciding on the type contract of `hypotenuse`:

- The sides of a triangle are measured with numbers. What kind of numbers, `int~s, ~floats`?
- The return value is also a number. Is the return type the same type as the parameters?

Since integer values can be represented as ~float~s, we settle on the this:

```
def hypotenuse(a: float, b: float) -> float:
```

The type contract says: if you pass two values of type `float` in your call to `hypotenuse`, the function will return a value of type `float`.

Hypotenuse Function Description

The function description states what the functions does. We place this description in the function's docstring. Any string that occurs as the first item in the definition of a module, function, class, or method is a docstring. By convention we use triple double quotes for docstrings.

```python
def hypotenuse(a: float, b: float) -> float:
    """Take the lengths of the two legs, a and b, of a right triangle
    and return the length of the hypotenuse.
    """
```

This incomplete but legal version of the function returns `None` because it doesn't have a return statement.

- Tip: We can stub the function with a return statement that returns a dummy value, like 0.0, so code that uses our function will work but produce incorrect results. That way we can get the "plumbing" of our program working before filling in the details of the functions.

Designing a Function Body

The function body implements an algorithm that produces the functions output (or effect) based on the function's inputs. The algorithm for calculating a hypotenuse is:

1. Square leg a
2. Square leg b
3. Sum the squares
4. Take the square root of the sum of the squares.

The last step produces the final result.

Later in the course will learn how to design algorithms. FOr now we can think of algorithm design intuitively.

The next slide shows the algorithm above translated to Python code.

Hypotenuse Function Body

```python
import math

def hypotenuse(a: float, b: float) -> float:
    """Take the lengths of the two legs, a and b, of a right triangle
    and return the length of the hypotenuse.
    """
    a2 = a * a
    b2 = b * b
    sum_squares = a2 + b2
    result = math.sqrt(sum_squares)
    return result
```

Of course this function can be shortened, but this version shows every detail.

Testing the Hypotenuse Function

We can test our function manually in the REPL or by adding example functions calls to a script. We should also add the examples we created in step 1 of the function design recipe to the docstring.

```python
import math

def hypotenuse(a: float, b: float) -> float:
    """Take the lengths of the two legs, a and b, of a right triangle
    and return the length of the hypotenuse.

    >>> hypotenuse(3, 4)
    5
    >>> hypotenuse(5, 12)
    13
    """
    a2 = a * a
    b2 = b * b
    sum_squares = a2 + b2
    result = math.sqrt(sum_squares)
    return result
```

If we do this then we get automated testing for free with doctest.

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'}
```

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.