A Short and Incomplete Introduction to Python

Part 2: Functions

Sigve Haug <sigve.haug@math.unibe.ch>,
Alexander Kashev <alexander.kashev@math.unibe.ch>
Science IT Support (ScITS), University of Bern

Based on a course by Riccardo Murri / Sergio Maffioletti
S3IT: Services and Support for Science IT, UZH

August 20–21, 2018

Functions

Functions, I

Functions are portions of the code given a name, which can be used to execute (**call**) them later.

Functions often take **arguments** — parametric values that can differ from one call to another.

After running, a function can **return** a value, just like using a variable to get its stored value.

Some functions are built into Python, like `print()`, and some come from **modules**.

Functions, II

Functions are called by adding a parenthesized argument list to the function name:

```
>>> int("42") # One argument, "42"
42
>>> int(4.2)
4
>>> float(42)
42.0
>>> str(42)
'42'
>>> str() # No arguments
''
>>> sum(2, 4) # Two arguments, 2 and 4
6
```

Functions, III

Some functions can take a variable number of arguments. For instance:

$\mathrm{sum}(x_0, \ldots, x_n)$ Return $x_0 + \cdots + x_n$.

$\max(x_0, \ldots, x_n)$ Return the maximum of $\{x_0, \ldots, x_n\}$

$\min(x_0, \ldots, x_n)$ Return the minimum of $\{x_0, \ldots, x_n\}$

Examples:

```
In [1]: min(1,2,3)
Out[1]: 1
In [2]: max(1,2)
Out[2]: 2
```

The most important function of all

help(`fn`) Display help on the function named `fn`

Exercise 2.A: What happens if you type these at the prompt?

- `help(abs)`
- `help(help)`

How to define new functions

The **def** statement starts a function definition.

```
def greet(name):
    """
    A friendly function.
    """
    print("Hello, " + name + "!")

# the customary greeting
greet("world")
```

```
def greet(name):
    """
    A friendly function.
    """
    print("Hello, " + name + "!")

# the customary greeting
greet("world")
```

Indentation is significant in Python: it is used to delimit blocks of code, like '{' and '}' in Java and C.

```
def greet(name):
    """
    A friendly function.
    """
    print("Hello, " + name + "!")

# the customary greeting
greet("world")
```

(This is a comment. It is ignored by Python, just like blank lines.)

This calls the function just defined.

```
def greet(name):
    """
    A friendly function.
    """
    print("Hello, " + name + "!")

# the customary greeting
greet("world")
```

What is this? The answer in the next exercise!

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

Exercise 2.B: Type and run the code on the previous page at the interactive prompt. (Pay attention to indentation!)

What's the result of evaluating the function `greet("world")`?

What does `help(greet)` output?

Default values

Function arguments can have default values.

```
>>> def hello(name='world'):
...     print ("Hello, " + name)
...
>>> hello()
'Hello, world'
```

Named arguments

Python allows calling a function with named arguments:

```
hello(name="Alice")
```

When passing arguments by name, they can be passed in any order:

```
>>> from fractions import Fraction
>>> Fraction(numerator=1, denominator=2)
Fraction(1, 2)
>>> Fraction(denominator=2, numerator=1)
Fraction(1, 2)
```

The ‘return‘ statement

```
def double(x):
    return x+x

double(3) == 6
```

The result of a function evaluation is set by the *return* statement.

If no `return` is present, the function returns the special value `None`.

```
def double(x):
    return x+x
    # the following line
    # is never exec'd
    print('Hello')
```

After executing `return` the control flow leaves the function.

Basic control flow

Control flow

When Python executes code, it goes from line to line, executing them in order.

We've already seen an exception to it: functions. The block of code they represent is executed when a function is called from other code.

```
def example():
  second() # Executes second

first()     # Executes first
example()   # Calls the example function
third()     # Executes last
```

Sometimes, we want **conditional** execution: some block of code is only run when some condition is met.

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
  # indented block
elif other-expr:
  # indented block
else:
  # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

Q: *Where's the 'end if'?*

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
  # indented block
elif other-expr:
  # indented block
else:
  # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

> **Q:** *Where's the 'end if'?*
> *There's no 'end if': indentation delimits blocks!*

Truth values

Any value can be tested in a condition, implicitly converting it to a boolean (`True`/`False`):

```
if 4: # Same as "if bool(4):"
  print("4 is considered 'true'")
```

A non-zero number, a non-empty string, a non-empty sequence (see next part) are all example of "truthy" values.

The number 0, the empty string, an empty sequence and `None` are "false".

```
if x != 0:
  print("x is not zero")

if x:
  print("x is not zero (if it's a number)")
```

`while`-Loops

Loops are blocks of code that execute repeatedly, usually until some condition is met.

Conditional looping uses the `while` statement:

```
while expr:
  # indented block
```

To break out of a `while` loop, use the `break` statement.

Use the `continue` statement anywhere in the indented block to jump back to the `while` statement.

You will see a more common form of loop, `for..in`, in the next part.

Exercise 2.C: Modify the `greet()` function to print "Welcome back!" if the argument `name` is the string `'Python'`.

Modules

Modules, I

The `import` statement reads a `.py` file, executes it, and makes its contents available to the current program.

```
>>> import hello
Hello, world!
```

Modules are only read once, no matter how many times an `import` statement is issued.

```
>>> import hello
Hello, world!
>>> import hello
>>> import hello
```

Modules, II

Modules are *namespaces:* functions and variables defined in a module must be prefixed with the module name when used in other modules:

```
>>> hello.greet("Python")
Hello, Python!
```

To import definitions into the current namespace, use the '`from` *x* `import` *y*' form:

```
>>> from hello import greet
>>> greet("Python")
Hello, Python!
```