University of Zurich[UZH]

Grid Computing Competence Center

Python basics

GC3: Grid Computing Competence Center,
University of Zurich

Sep. 28, 2012

The Python shell, I

Python is an *interpreted* language.

It also features an interactive "shell" for evaluating expressions and statements immediately.

The Python shell is started by invoking the command `python` in a terminal window.

```
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:13:53)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

The Python shell, II

Expressions can be entered at the Python shell prompt `>>>`; they are evaluated and the result is printed:

```
>>> 2+2
4
```

A line can be continued onto the next by ending it with the character '\'

```
>>> "hello" + \
... " world!"
'hello world!'
```

The prompt changes to '...' on continuation lines.

Reference:

http://docs.python.org/reference/lexical_analysis.html#line-structure

Basic types

Basic object types in Python:

- **bool** The class of the two boolean constants `True`, `False`.
- **int** Integer numbers: `1`, `-2`, ...
- **float** Double precision floating-point numbers, e.g.: `3.1415`, `-1e-3`.
- **str** Strings of byte-size characters.
- **list** Mutable list of Python objects
- **dict** Key/value mapping

The `list` and `dict` types are essential data structures, so we are covering them extensively afterwards.

String literals, I

There are several ways to express string literals in Python.

Single and double quotes can be used interchangeably:

```
>>> "a string" == 'a string'
True
```

You can use the single quotes inside double-quoted strings, and viceversa:

```
>>> a = "Isn't it ok?"
>>> b = '"Yes", he said.'
```

String literals, II

Multi-line strings are delimited by three quote characters.

```
>>> a = """This is a string,
... that extends over more
... than one line.
... """
```

(In other words, you need not use the \'s at the end of the lines.)

Operators

All the usual unary and binary arithmetic operators are defined in Python: `+`, `-`, `*`, `/`, `**` (exponentiation), `<<`, `>>`, etc.

Logical operators are expressed using plain English words: `and`, `or`, `not`.

Numerical and string comparison also follows the usual notation: `<`, `>`, `<=`, `==`, `!=`, ...

Reference:

- http://docs.python.org/library/stdtypes.html#boolean-operations-and-or-not
- http://docs.python.org/library/stdtypes.html#comparisons

Operators, II

Some operators are defined for non-numeric types:

```
>>> "U" + 'ZH'
'UZH'
```

Some support operands of mixed type:

```
>>> "a" * 2
'aa'
>>> 2 * "a"
'aa'
```

Some do not:

```
>>> "aaa" / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Operators, III

The "%" operator computes the remainder of integer division.

But it is also used for *string formatting*:

```
>>> "This is slide %d of %d" % (13, 32)
'This is slide 13 of 32.'
>>> "We are %.1f%% done." % (100.0 * 13/32)
'We are 40.6% done.'
>>> "Today is %s %d, %d" % ('October', 28, 2012)
'Today is October 28, 2012'
```

Reference: http://docs.python.org/library/stdtypes.html#string-formatting-operations

Expressions

Expressions are combinations of operations that manipulate values and return some other values. (Function calls are operations, too.)

For instance, `2+2` is an expression, as are `abs(-2)`, `os.path.exists('/tmp'), 1 + (1.0/2) + 2**(-2)`

Not all Python constructs return value. *Assignment, for example, does not:*

```
>>> a = 1
```

Similarly for the update operators +=, -=, etc.

References: http://lambda-the-ultimate.org/node/1044#comment-10878
http://docs.python.org/reference/expressions.html

Assignment, I

Assignment is done via the '=' statement:

```
>>> a = 1
>>> print a
1
```

There are a few shortcut notations:

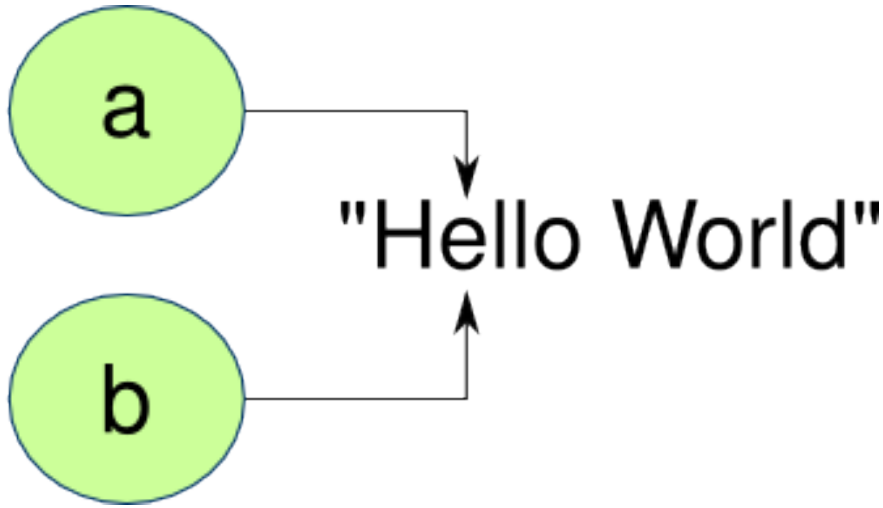
- `a += b` short for `a = a + b`,
- `a -= b` short for `a = a - b`,
- `a *= b` short for `a = a * b`,

etc. — one for every legal operator.

Assignment, II

Python variables are just "names" given to values. This allows you to *reference* the string `'Python'` by the *name* `a`:

```
>>> a = "Hello World"
```




```
>>> b = a
```

The same object can be given many names!

The `is` operator

The `is` operator allows you to test whether two names refer to the same object:

```
>>> a = 1
>>> b = 1
>>> a is b
True
```

Functions, I

Functions are called by postfixing the function name with a parenthesized argument list.

```
>>> int("42")
42
>>> int(4.2)
4
>>> float(42)
42.0
>>> str(42)
'42'
>>> str()
''
```

▸ More on int, float, str

Functions, II

Some functions can take a variable number of arguments:

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

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

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

Examples:

```
>>> sum(1,2,3)
6
>>> max(1,2)
2
```

The most important function of all

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

Question: *What happens if you type these at the prompt?*

- `>>> help(abs)`
- `>>> help(max)`

The most important function of all, II

When called without any argument, **help()** starts an interactive help prompt.

```
>>> help()
Welcome to Python 2.7! This is the online help utility.
If this is your first time using Python, you should definitely check out the tutorial
on the Internet at http://docs.python.org/tutorial/.
Enter the name of any module, keyword, or topic to get help on writing Python programs
and using Python modules. To quit this help utility and return to the interpreter,
just type "quit".
To get a list of available modules, keywords, or topics, type "modules", "keywords",
or "topics". Each module also comes with a one-line summary of what it does; to list
the modules whose summaries contain a given word such as "spam", type "modules spam".

help>
```

To return to the normal prompt, type `quit`

help('topic') has the same effect as typing `topic` at the interactive help prompt.

How to define new functions

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

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

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

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

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

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

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

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

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

This calls the function just defined.

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

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

What is this? The answer in the next exercise!

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

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

Exercise A: Type and run the code on the previous page at the interactive prompt. (Type indentation spaces, too!)
What does `help(hello)` print? What's the result of evaluating the function `hello("world")`?

Exercise B: Type the same code in a file named `hello.py`, then type `import hello` at the interactive prompt. What happens?

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.

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.hello("Bob")
Hello, Bob!
```

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

```
>>> from fractions import Fraction
```

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.

Question: *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.

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

Looping

Conditional looping uses the `while` statement:

```
while expr:
  # indented block
else:
  # executed at natural end of the loop
```

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

If a loop is exited via a `break` statement, the `else` clause is *not* executed.

Exercise C: Modify the `hello()` function to print "Welcome back!" if the argument `name` is your name.

Type conversions

str(x) Converts the argument x to a string; for numbers, the base 10 representation is used.

int(x) Converts its argument x (a number or a string) to an integer; if x is a a floating-point literal, decimal digits are truncated.

float(x) Converts its argument x (a number or a string) to a floating-point number.

▸ Back to Functions, I