Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 2
Hello, Python

The Python Interpreter, The Shell, Values, Operators, Expressions, Types, The Memory Model

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

Copyright © 2018 Kurt Rinnert, Kate Shaw
Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved. This file is offered as-is, without any warranty.

Abstract

"Programming is learned by writing programs."
– Brian Kernighan

We briefly describe how a computer runs programs, in particular Python programs.
We will describe the simplest Python statements and show how they can be used to do arithmetic. This is one of the most common tasks for computers and a great place to start learning to program.
We then introduce the Python memory model and the concept of a statement using the example of the assignment statement.
The concepts introduced here form the basis of everything that follows.

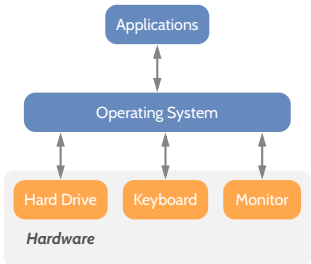
Overview

- How a computer runs Python programs
- The Python shell
- Values, operators, expressions
- Types
- Variables and computer memory

The concepts are not complicated but very important.

Computers & Operating Systems

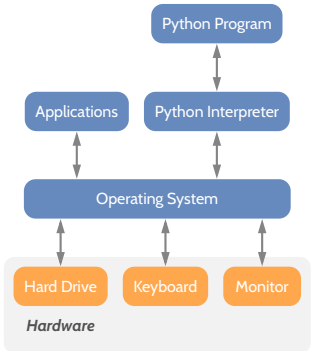
- A computer is built from pieces of hardware
- A special program provides access to the hardware
- This program is an *operating system*, or OS
- For example, Windows, OS X, Linux
- *Applications* interact with the OS to access the hardware
- Application programmers don't need to know hardware details as the OS handles them



Programmers need a basic understanding of the machines they use.

How Python Programs Run

- Python is a *high level* programming language
- The Python *interpreter* runs your Python programs
- The interpreter provides a *run time environment*
- This frees Python programmers (you!) from many headaches
- The interpreter is just another program
- That's right, your program runs in a program that runs in a program…
- Layers of abstraction like this are common



The machine you need to know about is a *virtual machine*: the Python interpreter.

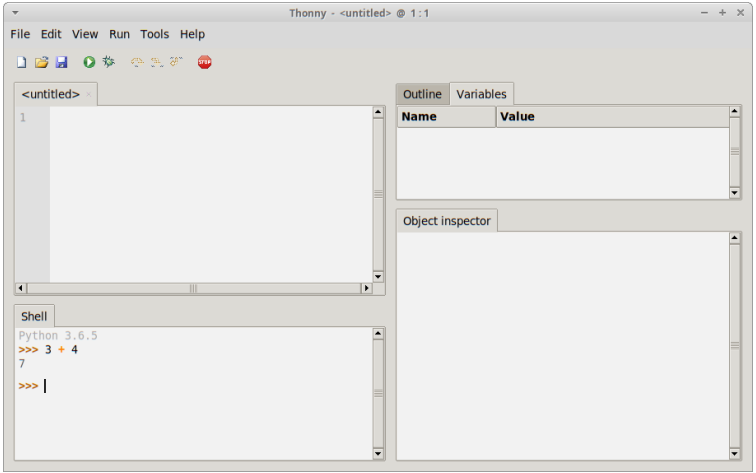
Expressions, Values, Operators

- You know mathematical *expressions* like 3 + 4
- Each expression is built from *values* like 2, 3 and 5
  and *operators* like + and —
- The operators combine their *operands* in different ways
- In the expression 4/5, the operator is "/" and the operands are 4 and 5
- Expressions can come without operators; the value 212 is an expression in itself
- Like any programming language, Python can *evaluate* basic mathematical expressions
- For example:

```
>>> 4 + 13
17
```

Expressions are the smallest building blocks of programs.

Try this in the IDE Shell



You can try any expression in the IDE shell.

Integer Division & Modulo

- Programmers often want to know the integer part of a division result
- For example, the number of days in 57 hours:

```
>>> 57 // 24
2
```

- We also often need the *remainder* of a division result:

```
>>> 57 % 24
9
```

- Beware of negative operands, the results might be surprising:

```
>>> -13 // 10
-2
>>> -13 % 10
7
>>> 13 % -10
-7
```

Make sure you understand these very useful operators.

Arithmetic Operators

Symbol	Operator	Example	Result
`-`	Negation	`-5`	`-5`
`+`	Addition	`11+3.1`	`14.1`
`-`	Subtraction	`5-19`	`-14`
`*`	Multiplication	`8.5*4`	`34.0`
`/`	Division	`11/2`	`5.5`
`//`	Integer Division	`11//2`	`5`
`%`	Remainder	`8.5%3.5`	`1.5`
`**`	Exponentiation	`2**5`	`32`

Familiarize yourself with operators on the Python shell.

What is a Type?

- We have seen two types of numbers, integers and floating-point
- We have also seen operators that can be used on them

Types are defined by:

1. A set of values
2. A set of operations that can be applied to the values

- We will soon see more types and later define our own

Programming is very much about understanding types.

Operator Precedence

- Let's say we want to convert Fahrenheit to Celsius:

```
>>> 212 - 32 * 5 / 9
194.22222222222223
```

- But the result should be `100.0`!
- This is because `*` and `/` have *higher precedence* than `-`.
- We have to use parentheses to get the correct result:

```
>>> (212 - 32) * 5 / 9
100.0
```

Make complicated expressions more readable with parentheses.

Operator Precedence

Precedence	Operator	Operation
highest	`**`	Exponentiation
	`-`	Negation
	`*`, `/`, `//`, `%`	Multiplication, Division, Integer Division, Remainder
lowest	`+`, `-`	Addition, Subtraction

Do not show off your knowledge of operator precedence. Use parentheses.

Finite Numeric Precision

- Values of type `float` are *not* exact fractions:

```
>>> 2 / 3 + 1
1.6666666666666665
>>> 5 / 3
1.6666666666666667
```

- The *rounding errors* look small
- But they can pile up in a long calculation
- Adding small `float` numbers to large `float` numbers can have no effect at all:

```
>>> 10000000000 + 0.00000000001
10000000000.0
```

There are entire books on the topic of numerical computation.

Variables: Remembering Values

- A variable is created by *assigning* it a value:

```
>>> degrees_celsius = 26.0
```

- This makes `degrees_celsius` refer to `26.0`:

```
>>> degrees_celsius
26.0
>>> 9 / 5 * degrees_celsius + 32
78.80000000000001
>>> degrees_celsius / degrees_celsius
1.0
```

- You can assign a new value to a variable (this does *not* create a new variable):

```
>>> degrees_celsius = 0.0
>>> 9 / 5 * degrees_celsius + 32
32.0
```

Variables refer to objects of a certain type.

Warning: = is *not* equality in Python!

- The meaning of = in Python is *not* the same as in mathematics.
- The assignment operator is not symmetric.
- n = 13 assigns the value 13 to the variable n.
- 13 = n results in an error.
- We read n = 13 as "n is assigned 13" rather than "n equals 13".

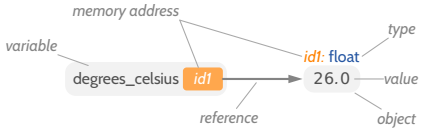
The Memory Model: Graphical Notation

- A floating point *value* `26.0` at the *memory address* `id1` is drawn like this:

- Note that we show the *type*, in this case `float`
- A drawing like this represents an *object*
- In our memory model, a *variable* contains the *memory address* of the *object* to which it *refers*
- We will draw arrows from *variables* to the *objects* they *refer* to:

You have to have a mental model of what's going on in the Python interpreter.

The Memory Model: Terminology



- The *value* `26.0` has the *memory address* `id1`
- The *object* at the *memory address* `id1` has the *type* `float` and the *value* `26.0`
- The *variable* `degrees_celsius` *contains* the *memory address* `id1`
- The *variable* `degrees_celsius` *refers* to the *object* of *type* `float` with the *value* `26.0`
- We also say the *variable is a reference to the object*

Programmers *should talk* about things *precisely*.

The Assignment Statement

- General form of an assignment statement:

```
variable = expression
```

- It is executed as follows:
  1. Evaluate the expression on the right of the = sign to produce a value.
     This value has a memory address.
  2. Store the memory address in the variable on the left of the = sign.
     Create a new variable if necessary.
     Otherwise just replace the memory address the variable contains.

The assignment makes the variable refer to an object.

Assignment Example

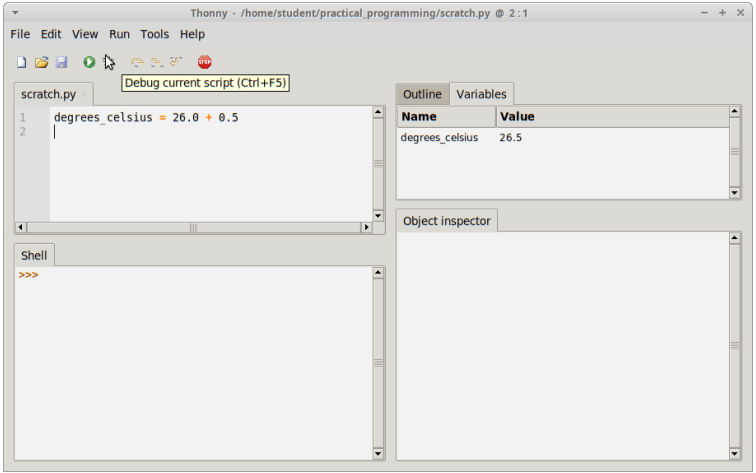
```
>>> degrees_celsius = 26.0 + 5
>>> degrees_celsius
31.0
```

```
degrees_celsius = 26.0 + 5
```

- The statement `degrees_celsius = 26.0 + 5` is executed as follows:
  1. Evaluate the expression on the right of the `=` sign: `26.0 + 5`.
     This produces the value `31.0`, which has a memory address.
  2. Make the variable on the left of the `=` sign, `degrees_celsius`,
     refer to `31.0` by storing the memory address of `31.0` in `degrees_celsius`.

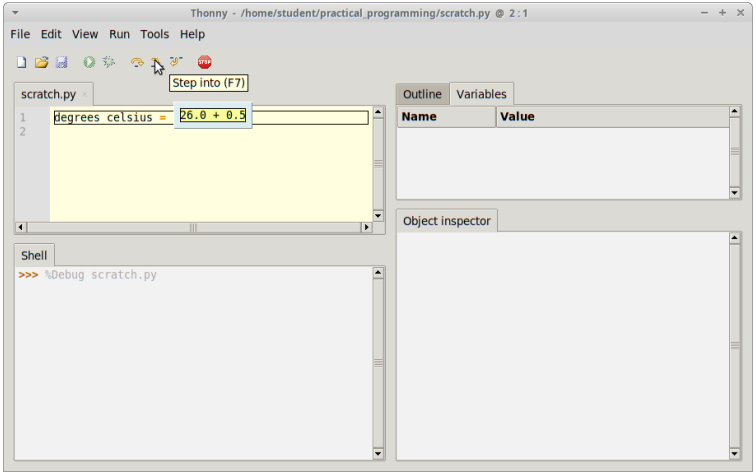
This time, the right hand side is not just a constant.

Try this in the IDE Debugger (Examination Mode)



Start stepping through by clicking the debug symbol.

Try this in the IDE Debugger (Examination Mode)



Watch what the Python interpreter does while stepping through.

Reassigning to Variables

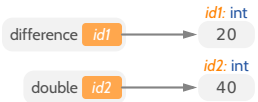
```
>>> difference = 20
>>> double = 2 * difference
>>> double
40
>>> difference = 5
>>> double
40
```

- First statement, `difference = 20`:
  1. Evaluate the expression on the right of the = sign: `20`.
     This produces the value `20`, which is put at the address `id1`.
  2. Make the variable on the left of the = sign, `difference`, refer to `20` by storing `id1` in `difference`.

So far just a simple assignment.

Reassigning to Variables

```
>>> difference = 20
>>> double = 2 * difference
>>> double
40
>>> difference = 5
>>> double
40
```

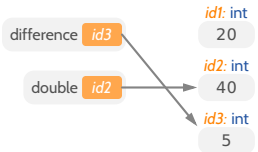


- Second statement, `double = 2 * difference`:
  1. Evaluate the expression on the right of the = sign.
     Because `difference` refers to the value `20`, this expression yields `2 * 20`.
     The resulting value `40` is stored at the memory address `id2`.
  2. Make the variable on the left of the = sign, `double`, refer to `40`
     by storing `id2` in `double`.

To evaluate the expression we *dereferenced* (accessed, used) a variable.

Reassigning to Variables

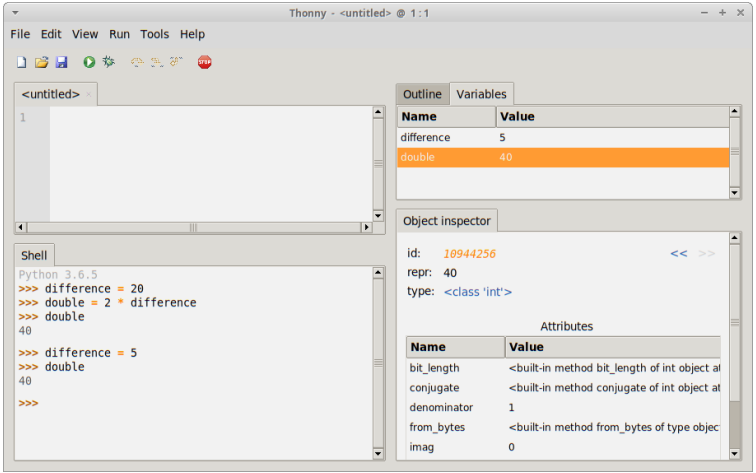
```
>>> difference = 20
>>> double = 2 * difference
>>> double
40
>>> difference = 5
>>> double
40
```



- Third statement, `difference = 5`:
  1. Evaluate the expression on the right of the = sign: 5.
     This produces the value 5 which is stored at the memory address `id3`.
  2. Make the variable on the left of the = sign, `difference`, refer to 5 by storing `id3` in `difference`.

The variable `double` did *not* change. No variable refers to `20` anymore.

Try this in the IDE Shell



Note how the values change in the variable inspector.

Augmented Assigment Operators

Symbol	Example (x = 5)	Result
`+=`	`x += 3`	x refers to 8
`-=`	`x -= 3`	x refers to 2
`*=`	`x *= 3`	x refers to 15
`/=`	`x /= 2`	x refers to 2.5
`//=`	`x //= 2`	x refers to 2
`%=`	`x %= 2`	x refers to 1
`**=`	`x **= 2`	x refers to 25

This is mostly *syntactic sugar*. That said, it is often more readable.

Writing Readable Code

- The following two expressions are equivalent:

```
num_customers = num_customers + 1
num_customers += 1
```

- The second is easier to read
- We use *white space* to make our code more readable
- Parentheses, even when not necessary, make expressions more readable

```
f = (x + 3.0) * 2.5
g = f * (5 / 9)
```

- *Don't* write like this:

```
g = f*5/9
```

Code is read much more often than it is written.

Describing Code: Comments

- Programs are often complicated and quite long
- We use *comments* to describe code, making it easier to understand for future readers
- In Python, a comment starts with the character #
- Everything after # on a line is ignored by Python
- But hopefully not by programmers reading your code!

```
# Python ignores this
# convert 212 degrees Celsius to Fahrenheit
100.0
```

- You can assume your readers will be programmers
- *Don't* do this:

```
num_customers += 1 # increment number of customers
```

Writing useful comments is an art.

When Things Go Wrong

Things Can Go Wrong in Two Ways

1. You write something Python can't digest (syntax error)
2. Your solution does not work (semantic error)

```
>>> 17 +
  File "<stdin>", line 1
    17 +
       ^
SyntaxError: invalid syntax
```

```
>>> 13 + foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

The reported errors are also called *exceptions*. We will use them to ask for forgiveness later.

Exercises Lecture 2