# Imperative programming with Python

January 2012 project: Class #2

Facundo Carreiro

ILLC, University of Amsterdam

January 10th, 2012

## Boolean expressions

- *Booleans* are expressions that can be either true or false.
- The boolean type has two values: `True` and `False`.

```
>>> type(True)
<type 'bool'>
```

```
>>> type(False)
<type 'bool'>
```

- We use comparison operators to form basic expression, e.g.

```
>>> 5 == 5
True
```

```
>>> 'Hello' == 'hello'
False
```

- We have many other operators and we can use them with variables

```
x != y                # x is different to y
x > y                 # x is greater than y
x < y                 # x is less than y
x >= y                # x is greater than or equal to y
x <= y                # x is less than or equal to y
x in y                # y contains x (in 'some' sense)
```

## Boolean expressions

- We have *logical operators* to form complex expressions

```
not x                      # true iff x is false
x and y                    # true iff x is true and y is true
x or y                     # true iff x is true or y is true
```

- Some examples

```
not x or y
(not x) or y                         # true iff x implies y
```

```
(x or y) and not (x and y)           # exclusive or
```

```
>>> ('lala' in 'shalala') and len('two') == 3
True
```

## Strings: the basics

- A *string* is a sequence of ‘letters’.
- They are specified with single and double quotation marks.

```
>>> type('hey ho')
<type 'str'>
```

```
>>> type("let's go")
<type 'str'>
```

- Let’s see some *potential* operations

```
>>> print ('2' - '1')
```

```
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

## Strings: the basics

```
>>> print 'bat' + 'man'
```

```
batman
```

```
>>> print 'gabba '*2 + 'hey!'
```

```
gabba gabba hey!
```

- The `len(·)` function returns the length of a string

```
>>> len("Education is a right that should be" +
... ' supported by the government')
64
```

## Strings and numbers: conversion

- We may want to convert numbers to strings

```
>>> avg = calculate_average()
>>> type(avg)
<type 'int'>
>>> print 'The average is: ' + avg + ', congratulations!'
TypeError: cannot concatenate 'str' and 'int' objects
```

- The `str(·)` function returns the string representation of a number.

```
>>> print 'The average is: ' + str(avg) + ', congratulations!'
The average is: 9.5, congratulations!
```

- Conversely, `int(·)` and `float(·)` convert strings to numbers

```
>>> type(int('282'))
<type 'int'>
```

```
>>> type(float('5.5'))
<type 'float'>
```

## Keyboard input

- The `raw_input(·)` function lets the user input some text with the keyboard

```
>>> i = raw_input()
Hello, my dear program
>>> print i
Hello, my dear program
```

- You can use it with a message

```
>>> i = raw_input('Are you talking to me?')
Are you talking to me?Yes
>>> print i
Yes
```

## Flow control: conditional execution

- The simplest form to control the flow of the execution is the *conditional execution* with the `if` statement

```
if boolean_expression: body
```

- A small example

```
name = raw_input('Please insert your name: ')
amount = int(raw_input('How much will you donate? '))

if amount <= 0:
    print 'You should input a positive number!'
    blacklist(name)
    quit()

process_donation(name, amount)
```

- Watch out: The body of the `if` statement is delimited by either tabs or spaces. This is called *indentation*. Do *not* mix tabs and spaces!

## Flow control: alternative execution

- Execution of alternatives is controlled with the `else` statement

```
if boolean_expression:
    [some code block]
else:
    [some code block]
```

```
dividend = int(raw_input('Insert the dividend: '))
divisor = int(raw_input('Insert the divisor: '))

# check if the division yields an integer number
if dividend % divisor == 0:
    print 'The result is: ' + str(dividend / divisor)
else:
    print 'I\'m sorry, I can\'t do that.'
```

## Flow control: chained conditionals

- You can chain conditionals with the `elif` statement (which stands for ‘else if’)

```
if boolean_expression:
    [some code block]
elif boolean_expression:
    [some code block]
else:
    [some code block]
```

Let’s see an example of all of them…

## Flow control: chained conditionals (example)

```
correct_answer = 762057
answer = input("What's the num of inhabitants in Amsterdam? ")

# compute the absolute distance to the correct answer
difference = abs(correct_answer - answer)

# ...
if answer < 0:
    print 'Are you insane?'
elif difference == 0:
    print 'Exactly!'
elif difference < 5000:
    print 'Quite close...'
elif difference < 50000:
    print 'You can do better!'
else:
    print 'Not even close...'
```

## Functions

- A *function* is a named sequence of statements that performs a computation.
- We have seen some functions already: `type(·)`, `abs(·)`, `int(·)`.
- A function is 'called' by its name and 'passing' some arguments separated by commas: $\texttt{name}(arg_1, \ldots, arg_n)$
- Calling a function temporarily *deviates* the flow of execution.
- The arguments can be values, variables, expressions.
- Functions can have a *return value*. For example we say that `abs(·)` takes a number as an argument and returns the absolute value.

## Functions and modules

- Functions have to be *defined* before they are used.
- `abs(·)`, `int(·)` are *built-in* functions, they are defined for you with the rest of the Python language.
- Tip: If you know the name of a function you can use the `help(·)` command to get the documentation about it

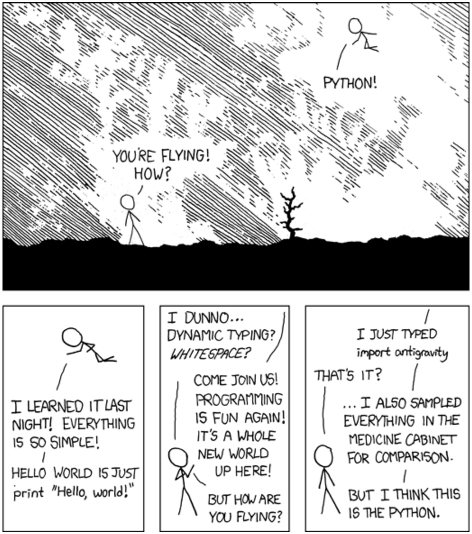
```
>>> help(abs)
abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

- In general, programming languages come with a *library* of functions organized in some way.
- In Python, the library is organized in *modules*.
- For the moment, a module is a collection of related functions.

## Python module library

By the XKCD webcomic


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

## Using modules

- As an example we will use the `random` module. It contains functions to generate random numbers in various probability distributions.
- First we need to *import* the module

```
>>> import random
```

- The functions in the module will be inside the random *namespace*. They are accessed using the *dot notation*

```
# Returns an integer from 1 to 10, endpoints included
>>> random.randint(1, 10)
7
```

- Suggested homework: read the book's intro to the `math` module.

## Using modules

- You can import functions into the main namespace

```
>>> from random import choice
>>> choice('abcdef')
c
```

- You can also import *everything* into the main namespace

```
>>> from random import *
```

- But please don't! unless it is extremely necessary.
- You can import modules and assign them a different name

```
>>> import random as r
```

## Why functions and modules?

1. Organization
   - Divide and conquer
   - Separation of concerns
2. Code reuse
   - Do not repeat yourself
   - Functions and modules can be shared among different programs
3. Maintainability
   - Easier to debug
   - Easier to read
4. Design for change
   - Define (or at least have in mind) an *interface* for each function
   - Encapsulate things that could change
   - Good practice foundation: Information hiding (David Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules")

## Interfaces

The *interface* of a function is a summary of how it is used:

- What are the parameters?
- What does the function do? as opposed to *how*.
- What is the return value? which are the side-effects?

A popular method is that of *pre-conditions* and *post-conditions*.

- It specifies a contract between the caller and the function.
- The precondition has to be satisfied by the caller.
- The caller can assume the postcondition.
- Written in some formal language.

## Interfaces: an example

Suppose we want to specify the `sort` function which takes a list of numbers and orders them.

- `sort(L:[Int]) → res:[Int]`
- `pre: True`
- `post`
  1. `ordered:` $\forall i, j \in \{0, \dots, |L| - 1\}, i < j \Rightarrow res_i \leq res_j$
  2. `same list:`$\forall e \in L, e \in res \wedge \forall e \in res, e \in L$ (too weak!)

     $\forall e \in L, \mathtt{count}(res, e) = \mathtt{count}(L, e) \wedge$
     $\forall e \in res, \mathtt{count}(res, e) = \mathtt{count}(L, e)$
     where $\mathtt{count}(A, e) := |[i : i \in \{0, \dots, |A| - 1\}, A_i = e]|$

As you can see,

- It helps spot possible mistakes.
- We end up having an unambiguous specification.
- *It is hard work*, even for simple and small functions.

## References

- Chapters 3, 5 and 6 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- Boolean operations
  http://docs.python.org/reference/expressions.html#boolean-operations
- Python: Myths about indentation
  http://www.secnetix.de/olli/Python/block_indentation.hawk
- The Python Standard Library
  http://docs.python.org/library/
- The `random` module
  http://docs.python.org/library/random.html