Imperative programming with Python

January 2012 project: Class #1

Facundo Carreiro

ILLC, University of Amsterdam

January 9th, 2012

Organizational & administrative stuff

- Instructor: Facundo Carreiro (http://www.science.uva.nl/~facundo/).
- Teaching Assistant: Fabio Zanasi (lastname.firstname@gmail.com).
- 6 ECs.
- From January the 9th to February the 3rd.
- 4x2h sessions per week (3 classes and 1 tutorial).
- Evaluation
  1. 3 homework sets (one per week in the first 3 weeks).
  2. A final project (in groups of around 3–4 people).

Contents (roughly)

1. Basic object-oriented programming
   - Variables, expressions and statements.
   - Functions.
   - Execution flow control.
   - Iteration and repetition.
   - Data structures: Lists, dictionaries, tuples.
   - File Input/Output.
   - Classes, objects and inheritance.
2. Python specifics
   - Python modules.
   - A glimpse of the *huge* Python standard library.
3. A bit of (unavoidable) software engineering
   - Group work.
   - Time estimation, testing, debugging.

What is programming all about?

- *"Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do."* – *Fellows, M.R., and Parberry, I.*
- Computer science is about *problem solving*.
- The most important skill is knowing how to analyze a problem, break it into pieces and solve it efficiently.
- Examples of problems:
  1. I want to get to Amsterdam Central Station as fast as possible.
  2. I want to sort a deck of cards.
  3. I have a meeting with my supervisor tomorrow and I didn't do anything.
- We will concentrate on problems that **can** be solved with a computer.

Algorithms

- An *algorithm* is a finite sequence of steps to solve a particular problem.

Input: Flour, eggs, sugar, butter, milk
Output: Cake

1. Heat the oven.
2. Mix ingredients in a bowl.
3. Put mixture in a baking pan.
4. Bake for 50 minutes.
5. Keep baking until top springs back when touched.

Input: A list of numbers $L = n_1, \ldots, n_k$
Output: L ordered by $<$

1. For `current_pos` := 1 to k do
   - Look for the smallest $n_{current\_pos}, \ldots, n_k$.
   - Switch it with $n_{current\_pos}$

Input: Money
Output: Cake

1. Go to Albert Heijn.
2. Look for cake.
3. Buy cake.

Programming languages

- Formal, unambiguous languages to write algorithms.
- Can be classified in *low-level* (aka. "machine code") and *high-level*.

Low-level programming languages

- Use a *tiny* set of instructions that the computer's CPU understands.
- Mess directly with the computer's memory cells.
- Are *architecture-specific* (only work for one specific CPU).

```
; Author: Paul Hsieh
gcd:    neg     eax
        je      L3
L1:     neg     eax
        xchg    eax,edx
L2:     sub     eax,edx
        jg      L2
        jne     L1
L3:     add     eax,edx
        jne     L4
        inc     eax
L4:     ret
```

Figure: GCD for Intel x86

High-level programming languages

- Complex set of instructions and constructions.
- More similar to natural language.
- Easier to read and write. Shorter (in general). *Portable*.

```
int GCD(int a, int b)
{
    while(1)
    {
        a = a % b;
        if(a == 0)
            return b;

        b = b % a;
        if(b == 0)
            return a;
    }
}
```

Figure: GCD in C

How does the computer execute a program?

- Strictly speaking, the CPU only understands 'bytecode' (binary encoded instructions).
- Low-level languages get compiled to bytecode quite easily.

  LL source code → Compiler → Bytecode → CPU → Output
- High-level languages can also be compiled

  HL source code → Compiler → Bytecode → CPU → Output
- and some of them can be *interpreted*! (e.g., Python!)

  HL source code → Interpreter → Output

  An interpreter is a program that translates and executes the code live.

Time for some Python!

- Python is an *interpreted* high-level language.
- It is an *imperative* language but also has some functional features.
- Let's take a look at our first Python program

```
# greetings!
print "Hello World!"

# assign some variables
a = 2
b = 4
c = 3

# print the average
print (a + b + c)/3
```

1. Run the Python interpreter and write this program.
2. Write this code in a file and run it (`python -i code.py`).
3. Run the code with '`python code.py`'.

Values and types

- A program works by manipulating *value*s.
- Values are classified in *types*.

Value	**Type**
$3, 4, 1000$	Integer
`'a', "Hello World", '"Freedom"'`	String
3.141592654	Float
`True, False`	Boolean
$\vdots$	$\vdots$

Figure: Some built-in data types

The `type(·)` function tells you the type of an expression

```
>>> type(3 + 8)
<type 'int'>
```

```
>>> type('stupid example')
<type 'str'>
```

Variables

- *Variables* are names that we assign to values.
- The `=` sign links a value to a variable.
- In Python, variables are created when first assigned a value.

```
n = 3
welcome_message = "Welcome␣aboard!"
```

should be read as $n \mapsto 3$ and `welcome_message` $\mapsto$ `Welcome aboard`.

- Variables can later be used in other expressions

```
>>> print n + 7
10
>>> n = n + 1
>>> print n
4
```

- *Good practice tip*: use meaningful names for variables!
- Watch out: Some names are reserved and can not be used as variable names (aka. reserved keywords).

Numeric types: Integers, floating point and friends

- *Integers* are (relatively) small numbers without a decimal part.
- It is the default type when you write a number without a point.

```
>>> type(2000)
<type 'int'>
```

```
>>> type(-15)
<type 'int'>
```

- They are bounded

```
>>> type(1234567890123)
<type 'long'>
```

```
>>> type(-1234567890123)
<type 'long'>
```

Numeric types: Integers, floating point and friends

- *Long* numbers have "unlimited" precision.
- You specify them by using an `L` suffix.

```
>>> type(2000L)
<type 'long'>
```

- If a number is too big to be an Integer it is interpreted a as Long.

```
>>> type(12345678901)
<type 'long'>
```

- Note: In Python 3.x Integers and Longs have been unified.

Numeric types: Integers, floating point and friends

- *Floats* are an approximation to real numbers.
- You specify them using a dot in the number.

```
>>> type(15.0)
<type 'float'>
```

```
>>> type(-3.141592654)
<type 'float'>
```

- They are also bounded

```
>>> print 0.1234567890123456
0.123456789012
```

Numeric types: operations

- Operations between numeric types include, among others: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), remainder (`%`).
- They are used in *infix* notation: `arg` $\odot$ `arg` .

```
>>> 3 + 2 ** 5 * 4 / 2
```

- Watch out for the precedence in the operations! The expression above is equivalent to...
  1. $3 + 2^{\frac{5 \times 4}{2}}$
  2. $(3 + 2)^{\frac{5 \times 4}{2}}$
  3. $3 + 2^5 \times \frac{4}{2}$
- We use *parenthesis* to group the expressions

```
>>> 3 + (2 ** (5 * 4 / 2))
1027
```

```
>>> (3 + 2) ** (5 * 4 / 2)
9765625
```

Numeric types: operations

- The same symbol may refer to different functions, i.e:

```
>>> 5/2
2
```

in this case $/ : \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$ therefore the answer gets rounded to an integer.

- If we use floats we get what we expected

```
>>> 5.0/2.0
2.5
```

- Which division do we use in the following line? does this work?

```
>>> 5/2.0
```

Yes, in these cases the most general function is used, i.e.:
$/ : \text{Float} \times \text{Float} \rightarrow \text{Float}$

Debugging

- It is the process of finding and fixing errors in a computer program.

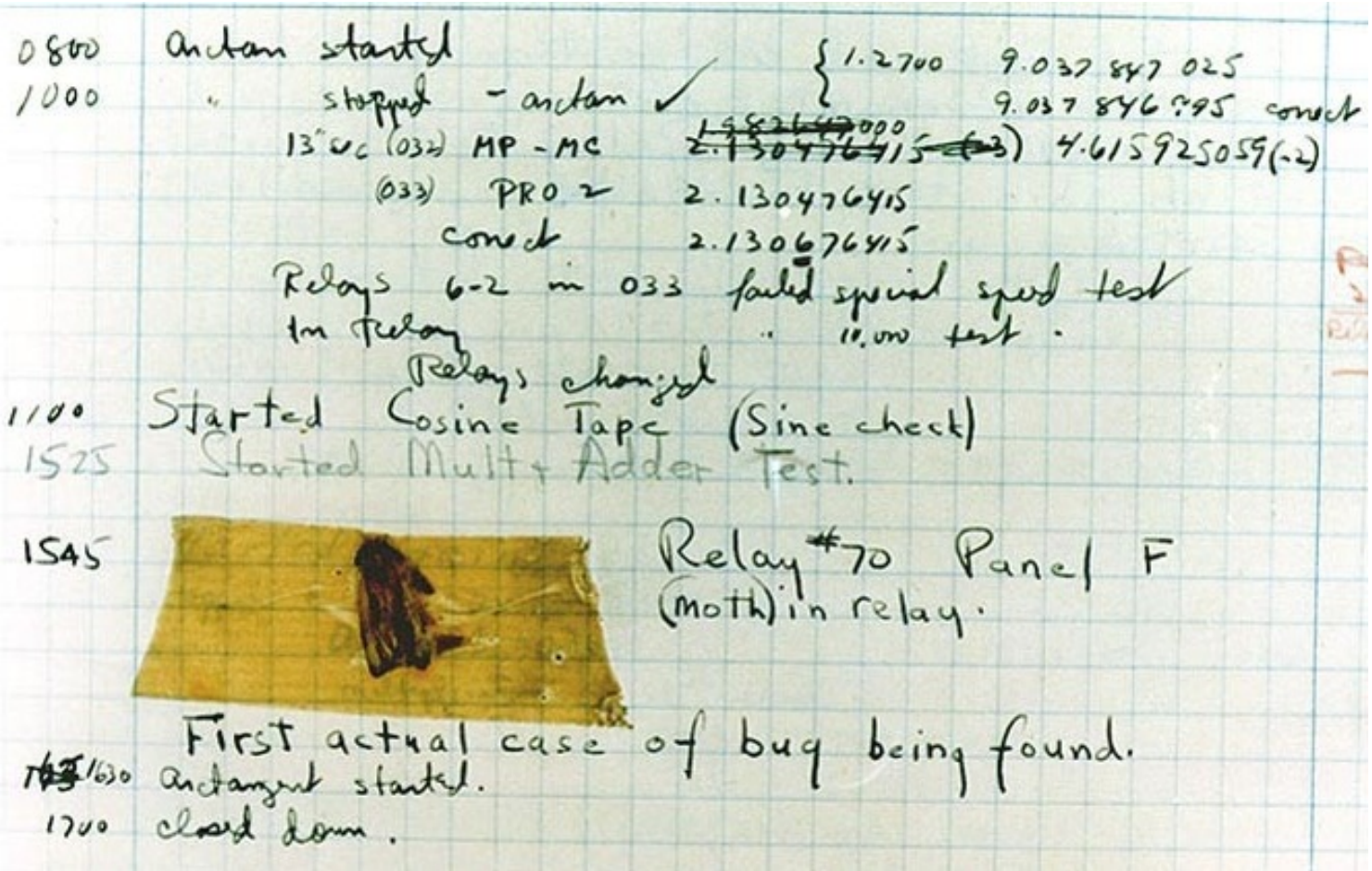

Figure: First computer bug (1947)

Debugging

- *Syntax errors*: Your text is not a valid Python program.

```
>>> n * 5 = 20
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

- *Runtime errors*: Found during the execution of a program.

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

```
>>> n = 0
>>> 5555/n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- *Semantic error*: No apparent error yet the program does not do what you want. May be cause, for instance, by
  - Order of operations
  - Wrong assumptions about an operation (e.g. integer division)

References

- Chapters 1 and 2 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- Python documentation (extremely useful)
  http://docs.python.org/
- Numeric types
  http://docs.python.org/library/stdtypes.html#typesnumeric
- Floating point
  http://docs.python.org/tutorial/floatingpoint.html
- First computer bug
  http://www.history.navy.mil/photos/images/h96000/h96566kc.htm