Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Acknowledgement: These course notes are based on a previous set of notes written by Vedran Šego (vsego.org (http://vsego.org/)) in 2015.

Contents: ¶

An algorithm

- *A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer*
  http://www.oxforddictionaries.com/definition/english/algorithm?q=algorithm (http://www.oxforddictionaries.com/definition/english/algorithm?q=algorithm)
- Informal definition from Wikipedia: *a set of rules that precisely defines a sequence of operations*.
- More precisely (albeit still a bit informal):
  *A **sequence** of actions that are always executed in a **finite** number of steps, used to solve a certain problem.*
- Simplified: a cookbook, often very general

Important parts of an algorithm:

1. get the input data,
2. solve the problem,
3. output the result(s).

Example: preparing a frozen pizza. Steps:

1. Get temperature *temp* and time *t* (written on the box).
2. Heat oven to the temperature *temp*.
3. Discard all packaging (recycle the box).
4. Cook for a time *t*.
5. Get pizza out of the oven.
   Use some heat-resistent hands protection or execute the "go to the ER" subroutine.

Example of the input data:

- *temp* = "conventional oven: 190C; fan oven: 180C",
- *T* = between 15 and 17 minutes.

A program

- *A series of coded software instructions to control the operation of a computer or other machine.*
  http://www.oxforddictionaries.com/definition/english/programme
  (http://www.oxforddictionaries.com/definition/english/programme)
- Simplified: precise instructions written in some programming language

Our focus

- **Algorithms,** not programs!
- We shall use programs to test algorithms and to see how they work.
- **Aim:** Learn to solve problems using a computer, **regardless of the choice of language** (which may or may not be Python in the courses you take in your future work or studies).

Types of programming languages (implementations)

Compiled

- Require compilation (translation) of the code to the machine code (popular, albeit a bit meaningless, "zeroes and ones"),
- Usually strictly typed,
- Usually faster than interpreters.

Interpreted

- Translated during the execution,
- Usually untyped or loosly typed,
- Usually slow(ish).

Where is Python?

- Simplified: an interpreter.
- Programs get semi-translated when run (*pseudocompiler*).
- Untyped, slower than compiled languages (but with various ways to speed up).

No details – too technical and they depend on the specific Python implementation.

Python versions

Many implementations, but two major versions

- Python 2
- Python 3 ⇜

Minor differences in what we need (these will be mentioned).

An example: Hello world

In [1]:

```
 rint("Hello, World!")
```

```
Hello, World!
```

The output can be nicely formated, but more on that in the near future.

What is "Hello World"?

A very simple program, used to show the basic syntax of a programming language.

See the The Hello World Collection (http://www.roesler-ac.de/wolfram/hello.htm) for the examples in many other programming languages.

The need for such an example can be clearly seen from the examples of more complex, but still fairly readable languages (for example, various versions of C++ and Java).
Beware of the Assembler-Z80-Console and BIT examples. ☺

Note: A full Python program should always start with the following:

- the first line: `#!/usr/bin/env python3`
  This has no meaning on Windows, but it makes your programs easier to run on Linux and Mac OSX.
- Right after that, a description what the program does, possibly with some other info (system requirements, authors name and contact, etc.), between the triple quotation marks. This is called a *docstring* and it will be further addressed next week.

So, a full "Hello World" program would look like this:

In [9]:

```
 !/usr/bin/env python3

"""
  program that prints the "Hello, World!" message.
"""

 rint("Hello, World!")
```

```
Hello, World!
```

We typically omit these elements in the lectures and we will mostly present chunks of code (that are not necessarily full programs) to save some space.

The following line that Spyder adds to new files

> `-*- coding: utf-8 -*-`

is not necessary in Python 3. However, if you are using international characters in some encoding other than UTF-8 (which you really shouldn't do!), this is a way to specify that encoding.

In this course we shall not cover encodings, as it is a very technical subject and most of the time it is enough to just use UTF-8 (find it in your editor's settings). However, if you're ever to build an application with the need for international characters, do look up the encodings on the internet (the Wikipedia page (http://en.wikipedia.org/wiki/Character_encoding) is a good start) and use UTF-8 (http://en.wikipedia.org/wiki/UTF-8) whenever possible, as it is a widely accepted standard and the default in Python 3. You should also make sure that your editor saves files using the UTF-8 encoding (Spyder does that by default).

In Python 2, the default encoding is ASCII (http://en.wikipedia.org/wiki/ASCII) and the UTF-8 support has to be enabled manually.

Comments

It is often useful to add human language notes in the code. These are called *comments* and are ignored by a computer, but they help programmers read the code.

In Python, comments are made by prepending the hash sign # in front of it. Each comments ends with the end of the line.

It is a standard to **always write comments in English:**

> Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language. Source: PEP 8 (https://www.python.org/dev/peps/pep-0008/#comments)

For example:

In [ ]:

```
 !/usr/bin/env python3

"""
  program that prints the "Hello, World!" message.
"""

  A welcome message
 rint("Hello, World!")
  TODO: ask user their name and save it as a file
```

As you can see, the above code runs just as the previous one, but a programmer reading it can get more information about the code itself.

Some editors will recognize certain *tags* in comments and highlight them to make them more noticable (like the `TODO` tag in the previous example). Some of the more common ones are, as listed in the Wikipedia's comments article (http://en.wikipedia.org/wiki/Comment_%28computer_programming%29#Tags):

- `FIXME` to mark potential problematic code that requires special attention and/or review.
- `NOTE` to document inner workings of code and indicate potential pitfalls.
- `TODO` to indicate planned enhancements.
- `XXX` to warn other programmers of problematic or misguiding code.

We shall often use the comments to denote what certain parts of the code do. These should always be descriptive and not merely rewritten code.

For example, this is good:

```
    Get the sum of primes in `L` as `prime_sum`
for x in L:
    if is_prime(x):
        prime_sum += x
```

as it makes clear what the code is doing, even to someone who doesn't "speak" Python.

This is bad:

```
    For each element `x` in the list, if `x` is a prime number,
    add it to `prime_sum`.
for x in L:
    if is_prime(x):
        prime_sum += x
```

because this comment is just a rewrite of the code that follows it and, as such, it is useless.

It is advisable to keep all lines (comments and docstrings) wrapped under 80 characters, although it shouldn't be forced when it reduces the code readability.

Input

How do we **input** some data?

Not surprisingly, using the function `input()`.

In [10]:

```
  = input()
rint("The value of x is", x)
```

```
17
The value of x is 17
```

Well, this `x` looks kind of important here. What could it be?

Variables

- Variables are used to store and retrieve data.
- Every variable has a value (which, in Python, *can* be undefined) and a type (partly hidden in Python).

Simple types

- **Number** -- generally, what you'd consider an integer or a real number in mathematics.
  For example: 17, -19, 17.19, 0, 2e3, ...
- **String** -- a text.
  For example: `"word"`, `"This is a mighty deep, philosophical sentence."`, `"ŞpęÇïåŁ sɹəɬɔɐɹɐɥɔ"`, "17", ...
  The so-called *empty string*, `""`, has no characters (its length is zero).
- **Boolean** -- truth values (`True` and `False`).
- **`NoneType`** -- the type of a special constant `None` that means "no value".
  This is **not** a zero (a number), nor an empty string, nor anything else! `None` is different from any other constant and any other value that a variable can get.

Be careful: 17 is a number, while `"17"` is a string!

Some not-so-simple types

- Lists and tuples -- most languages have only lists (usually called *arrays*)
- Dictionaries
- Sets
- Objects
- Functions (yes, functions can be saved in variables as well)
  ...

More on these in the near future.

How do variables work?

Let us analyze this piece of code:

In [12]:

```
  = input()
rint("The value of x is", x)
```

```
a mistery
The value of x is a mistery
```

Whatever is on the right-hand side of the assignment = gets computed first. Then the result is assigned to the variable on the left-hand side. When this is done, the next line of code is executed.

In our concrete example this means:

1. The function `input()` reads a sequence of characters from the standard input (usually the user's keyboard) and returns it as a **string**.
2. That value is then assigned to the variable `x` (on the left-hand side of the assignment operator =).
   Now `x` holds - as a string - whatever we have typed up to the first newline, i.e., up to the first Enter key (the newline itself is not part of the string).
3. The function `print()` now outputs its arguments to the standard output (usually the user's screen), in order in which they were given, separated by a single space character. So,
   - First, a string `"The value of x is"` is written out.
   - Then a singe space character is written out.
   - Then the value of `x` is written out (**not** the string "x" itself, because `x` is a variable!).

In other words, if we type "17", our program will write

```
he value of x is 17
```

And if we type "a mistery", our program will write

```
he value of x is a mistery
```

Don't expect Python to do anything *smart* here. It just writes out the values as they were given.

Python 2 remark: In Python 2, `print` does not need parentheses (i.e., `print "Hello, World!"` is fine).
However, do include them even if writing a Python 2 program, to make it easier to port to Python 3, and to avoid problems in some more advanced uses you might encounter in the future.

On the simple types

Be careful! Even if a value *looks* like a number, it *might* **not be** one!

Let us try to input two numbers in the following code, also adding the descriptions of what is expected in each of the inputs:

In [13]:

```
  = input("x: ")
  = input("y: ")
 rint(x, "+", y, "=", x+y)
```

```
x: 17
y: 19
17 + 19 = 1719
```

What did just happen here?

The user types two numbers, which are saved -- as two **strings** -- in variables `x` and `y`. Then the program writes out (among other things) the value of `x+y`.

How would we "add" one string to another in the real world?

For example, if `x = "Bruce"` and `y = "Wayne"`, what would `x + y` be?

(It may come as a little surprise that `"x + y"` will **not** produce `"Batman"`. Python is a well defined language that keeps Bruce's secret identity well hidden.)

The result of x+y will be `"BruceWayne"`. Notice that there is no additional space here: the strings are *glued* (concatenated) one to another, with no extra separators!

So, what happens if `x = "17"` and `y = "19"`?

It would be very bad if Python looked at these and decided that they were numbers only because they have nothing but digits. Maybe we want them concatenated (as opposed to adding them one to another)!

So, the result is -- by now no longer surprisingly -- "1719", because the strings' addition + is a concatenation, regardless of the value of the strings in question.

How do we tell Python to "treat these two variables as numbers"?

Converting basic types

We can explicitly tell Python to convert a string to an integer or a real number, and vice versa.

In [14]:

```
  = int(input())
  = float(input())
rint("x = ", x)
rint("y = ", y)
rint("x+y = ", x+y)
  = 'This is a string: "' + str(x+y) + '"'
rint(z)
```

```
17
1.23
x =  17
y =  1.23
x+y =  18.23
This is a string: "18.23"
```

We see three conversion functions:

- `int()`, which takes a string and converts it to an integer. If the argument is not a string representation of an integer, an error occurs.
- `float()`, which takes a string and converts it to a "real" number (also called ***float****ing point number*, hence the name of the function). If the argument is not a string representation of a real number, an error occurs.
- `str()`, which takes a number (among other allowed types) and converts it to a string.

Python 2 remark: In Python 2, `input()` is similar to `float(input())` in Python 3 (actually, `eval(input())`, but that is well beyond this lecture). This means it loads a number and returns it as a floating point number (causing an error or a strange behaviour if anything else is given as the input).
To load a string in Python 2, one has to call `raw_input()` (which does not exist in Python 3).

A note on the string creation: There are better ways to form the variable `z` (using various *formatting* methods), but this will have to wait a few weeks until we cover strings in more depth than here.

More on the assignments

What will the following code print?

In [15]:

```
 = 17
rint("The value of x was", x)
 = x + 2
rint("The value of x is", x)
```

```
The value of x was 17
The value of x is 19
```

As we said before: whatever is on the right hand side of the assignment `=`, gets computed **first**. Only **after that**, the result is assigned to the variable on the left hand side.

So, when Python encounters the command

```
    = x + 2
```

while the value of `x` is 17, it first computes `x + 2`, which is 19. After that, it performs the assignment `x = 19`, so 19 becomes the **new value** of `x` (which is then displayed with the second `print` function).

In most of the modern languages, `x = x + y` can be written as `x += y`. The same shortcut works for other operators as well, i.e., `x = x op y` can be written as `x op= y`.

For basic numerical operations, this means we have the following shortcuts:

Expression	Shortcut
`x = x + y`	`x += y`
`x = x - y`	`x -= y`
`x = x * y`	`x *= y`
`x = x / y`	`x /= y`
`x = x // y`	`x //= y`
`x = x % y`	`x %= y`
`x = x ** y`	`x **= y`

A note on other languages: there are no increment (++) and decrement (`--`) operators in Python.

Some special operators

Most of the operators in the above table have the same meaning as in mathematics (**for those knowing C:** `/` means the *usual*, i.e., real division). The three not used in mathematics are defined as follows:

- `x // y` means floored quotient of x and y (also called *integer division*), i.e., `x // y` $:= \lfloor \mathrm{x} \,/\, \mathrm{y} \rfloor$,
- `x % y` means the remainder of x/y, i.e., `x % y := x - y * (x // y)`,
- `x ** y` means x^y (x to the power y).

Python 2 remark: In Python 2, the ordinary real division `x/y` works in a C-like manner, which means that `x/y` is equivalent to `x//y` if both x and y are integers.
In Python 3, `x/y` always means real division. In other words,

- Python 2: `3//2 = 3/2 = 1`, but `3/2.0 = 3.0 / 2 = 3.0 / 2.0 = 1.5`;
- Python 3: `3//2 = 1`, but `3/2 = 3/2.0 = 3.0 / 2 = 3.0 / 2.0 = 1.5`.

Real number trouble

When dealing with real numbers, one must be extremely careful!

Simple arithmetics

What happens when we try to compute `a + b - a` for several different real values of `a` and `b`? Fairly often, the result will **not** be `b`!

In [16]:

```
  = 10
  = 0.1
 rint("a =", a, "  b =", b, "  ->  ", "a + b - a =", a + b - a, "!=", b, "= b")
  = 10**7
  = 10**(-7)
 rint("a =", a, "  b =", b, "  ->  ", "a + b - a =", a + b - a, "!=", b, "= b")
  = 10**11
  = 10**(-11)
 rint("a =", a, "  b =", b, "  ->  ", "a + b - a =", a + b - a, "!=", b, "= b")
```

```
  = 10   b = 0.1   ->   a + b - a = 0.09999999999999964 != 0.1 = b
  = 10000000   b = 1e-07   ->   a + b - a = 1.0058283805847168e-07 !
= 1e-07 = b
  = 100000000000   b = 1e-11   ->   a + b - a = 0.0 != 1e-11 = b
```

A division

There is no such thing as a *real* number in a computer. All numbers are actually (something like) decimals with an upper limit on the number of correctly remembered digits. The rest of the digits is lost, which can produce weird results, like `x * (1 / x) ≠ 1`.

In [17]:

```
  = 474953
  = 1 / x
 rint(x * y)
```

```
0.9999999999999999
```

Use integers whenever possible

Fibonacci numbers are defined as follows:

$$F_0 := 0, \quad F_1 := 1, \quad F_{n+1} := F_n + F_{n-1}, \quad n \geq 1.$$

There is also a direct formula for computing F_n:

$$F_n = \frac{\varphi^n - \quad ^n}{\overline{5}}, \quad \varphi := \frac{1 + \quad \overline{5}}{2}, \qquad := \frac{1 - \quad \overline{5}}{2}.$$

Mathematically, both definitions are equivalent. On a computer, however, the second will soon give you wrong results.

In the following code, `fib1(n)` returns the n-th Fibonacci number computed by a simple integer-arithmetics algorithm, while `fib(2)` uses the above formula (never use the recursive definition for i i i !.

In [1]:

```
 ef fib1(n):
    f0 = 0
    f1 = 1
    while n > 1:
        (f0, f1) = (f1, f0 + f1)
        n -= 1
    return f1

 ef fib2(n):
    sqrt5 = 5 ** .5
    phi = (1 + sqrt5) / 2
    psi = (1 - sqrt5) / 2
    return int((phi**n - psi**n) / sqrt5)

  = int(input("Type n (try to go for 73 or more): "))
 ib1n = fib1(n)
 ib2n = fib2(n)
 rint("|fib1(n) - fib2(n)| = |" + str(fib1n), "-", str(fib2n) + "| =", abs(fib1n
- fib2n))
```

```
Type n (try to go for 73 or more): 100
|fib1(n) - fib2(n)| = |354224848179261915075 - 354224848179263111168
| = 1196093
```

Even a simple addition incurs errors

The following code computes and prints three sums:

$$\sum_{i=0}^{999} 0.1 = 100, \qquad \sum_{i=0}^{9999} 0.1 = 1000, \quad \text{and} \quad \sum_{i=0}^{9999999} 0.1 = 10^6.$$

In [21]:

```
 = 0
or _ in range(1000):
   s += 0.1
rint(s)
 = 0
or _ in range(10000):
   s += 0.1
rint(s)
 = 0
or _ in range(10000000):
   last = s
   s += 0.1
rint(s)
```

```
99.9999999999986
1000.0000000001588
999999.9998389754
```

Notice how the result is sometimes smaller and sometimes bigger than the correct result.

Associativity of addition

We all know that for a finite set of real numbers $\{a_1, \ldots, a_n\}$ the following is true:

$$\sum_{i=1}^{n} a_i = \sum_{i=n}^{1} a_i = \sum_{i=1}^{n} a_{P(i)},$$

for any permutation P. However, in a computer, this isn't always so.

In [2]:

```
 rom math import pi
  = 15 * pi
  Create the list of series elements
lts = [ ]
  = 1
or k in range(1, 150, 2):
   elts.append(x**k / f)
   f *= -(k+1) * (k+2)
  Sum elements in the original order
in1 = 0
or el in elts:
   sin1 += el
rint("sin1 =", sin1)
  Sum elements in the reversed order
in2 = 0
or el in reversed(elts):
   sin2 += el
rint("sin2 =", sin2)
  Sum elements from the middle one to the ones on the edges
nt = len(elts)
id = cnt // 2
in3 = 0
or i in range(mid + 1):
   if mid + i < cnt:
       sin3 += elts[mid + i]
   if i:
       sin3 += elts[mid - i]
rint("sin3 =", sin3)
  Sum elements from the ones on the edge to the middle one
in4 = 0
or i in reversed(range(mid + 1)):
   if mid + i < cnt:
       sin4 += elts[mid + i]
   if i:
       sin4 += elts[mid - i]
rint("sin4 =", sin4)
rint("|sin1 - sin4| =", abs(sin1 - sin4))
rint("the first element:", elts[0])
rint("the last element:", elts[-1])
```

```
sin1 = -3121.3699495895926
sin2 = -2947.8076865687467
sin3 = -2403.8076865683283
sin4 = -1768.0
|sin1 - sin4| = 1353.3699495895926
the first element: 47.12388980384689
the last element: 5.3966776616824465e-12
```

The above is the computation of $\sin(15\pi)$ via the first 74 elements of the Taylor series (http://en.wikipedia.org/wiki/Sine#Series_definition) of the sine function:

- `sin1` computation starting from the first element ($a_1 + a_2 + a_3 + \cdots$),
- `sin2` going from the last to the first element ($a_{74} + a_{73} + a_{72} + \cdots$),
- `sin3` going from the center out ($a_{37} + a_{36} + a_{38} + a_{35} + a_{39} + \cdots$),
- `sin4` going from the edges in ($a_1 + a_{74} + a_2 + a_{73} + \cdots$).

The difference between `sin1` and `sin4` is roughly 1353, which may not look like much, but it is far more than the difference between any two sines should be.

You might also notice that $\sin(15\pi)$ shouldn't be anywhere near -3000 or -1768.

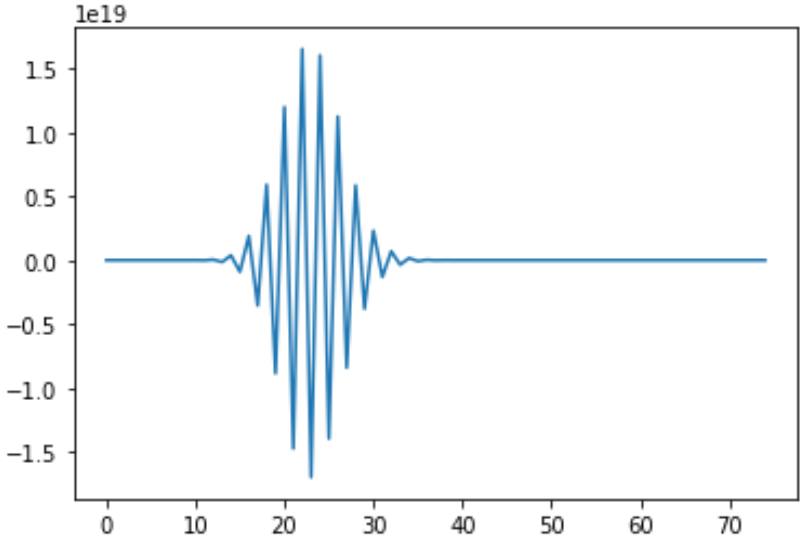
One might think that we should compute more elements of the sum, but this is not the case: the last element of the sum is only around $5.4 \cdot 10^{-12}$ (and the following ones would be even smaller).

So what happened here?

A detailed explanation is part of Numerical Analysis, but the key is in the largely varying magnitude and alternating signs of the elements:

In [3]:

```
matplotlib inline
mport matplotlib.pyplot as plt
lt.plot(elts)
lt.show()
```



Powers

Let us define

$$f(x,n) := \underbrace{\sqrt{\sqrt{\ldots\sqrt{x}}}}_{n}, \quad g(x,n) := \ldots(x)^{2^{2^{\overbrace{\ldots}^{n}}}}.$$

In other words, $f(x,n)$ is the number that we get by taking the square root of x, n times in a row, and $g(x,n)$ is the number we get by computing the second power of x, n times in a row.

Obviously, $x = f(g(x,n),n) = g(f(x,n),n)$ for any $n \in \mathbb{N}$ and $x \in \mathbb{R}_0^+$. But, let's see what a computer has to say if we input some $x \quad 1$ and $n = 50, 60, \ldots$:

In [24]:

```
rom math import sqrt
 = float(input("x = "))
 = int(input("n = "))
 = x
or _ in range(n):
   t = sqrt(t)
or _ in range(n):
   t *= t
rint("g(f(" + str(x) + ", " + str(n) + "), " + str(n) + ") =", t)
 = x
or _ in range(n):
   t *= t
or _ in range(n):
   t = sqrt(t)
rint("f(g(" + str(x) + ", " + str(n) + "), " + str(n) + ") =", t)
```

```
x = 1.7
n = 53
g(f(1.7, 53), 53) = 1.0
f(g(1.7, 53), 53) = inf
```

Do these tiny errors really matter?

Yes, rounding errors have repeatedly led to catastophic consequences, for example, in engineering, finance, and science. See [http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html (http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html)] for an interesting list. Even when solving a linear system of equations, probably the most fundamental problem in scientific computing, rounding errors have to be taken care of.

Consider the following two systems of linear equations:

$$\begin{aligned} 1 \cdot x &+ 1 \cdot y = 2, \\ 1.000001 \cdot x &+ 1 \cdot y = 2.000001, \end{aligned} \quad \text{and} \quad \begin{aligned} 1 \cdot x &+ 1 \cdot y = \\ 1.000001 \cdot x &+ 1 \cdot y = 1.9999 \end{aligned}$$

What are their solutions?

The solution to the first one is $(x, y) = (1, 1)$, but the solution to the second one is $(x, y) = (-1, 3)$.

Notice that both systems only differ by a tiny change of magnitude 10^{-6} in just one element, but their solutions (x, y) are completely different! Such a small change could easily be caused by one of the small errors shown before. Similar results can be achieved with arbitrarily small errors.

Bottom line

Always be extra careful when working with "real" numbers in a computer (or, better, avoid them altogether if possible, like in the Fibonacci example)!

These errors cannot always be considered insignificant, as they can pile up and/or grow in subsequent computations.

Without your lecturer being biased at all, anyone intending to do serious computations with computers should take the course "Numerical Analysis 1 (http://www.maths.manchester.ac.uk/study/undergraduate/courses/mathematics-bsc/course-unit-spec/?unitcode=MATH20602)" (MATH20602).