Basics I - Data Types

Gabriele Pompa
gabriele.pompa@unisi.com

March 23, 2020

Contents

Resources:

- *Python for Finance (2nd ed.)*: Sec. 3.Basic Data Types (Section 3.Excursus: Regular Expression is optional)
- *The Python Tutorial*: Sec. 3.1.1 (Numbers), 3.1.2 (Strings), 3.2 (First Steps Toward Programming), 4.1 (if Statements)

Executive Summary

Informally, the *type* of a variable is related to the amount of bits that are reserved in memory to store it.

The Python interpreter infers at run-time the type of a variable. Thus we say that Python is a *dynamically typed* language. This to contrast it with other - compiled - languages, like C or C++, where the type of a variable has to be declared when the variable identifier (that is, its name in our code) is introduced for the first time in the code. The latter kind of languages are told *statically typed* languages.

The function `type()` can be called over any defined variable and returns its type.

The following sections are organized as follows:

- In Section 1 we introduce integer numbers (or `int`), which are the Python data type to represent integers like 1, 2, 3,...

- In Section 2 we introduce float numbers (or `float`), which are the Python data type to represent fractions like 1/2, 0.25 or real numbers like e, π,...
- In Section 3 we introduce booleans (or `bool`), which are the Python data type to represent the logical values `True` or `False`. In this context `while` loops (in Sec. 3.1) and `if` statements (in Sec. 3.2) are introduced.
- In Sec. Section 4 we introduce strings (or `str`), which are the Python data type to represent text like `"this one"`.

1 Integers

Integers like 1, 2, 3,... are represented in Python as `int` data type.

[1]:
```
n = 10
type(n)
```

[1]:
```
int
```

The amount of bits (i.e. memory) reserved to an `int` depends on its value. For `n=10`, 4 bits are reserved. We can check this using the `.bit_length()` method of `int` variables.

[2]:
```
n.bit_length()
```

[2]:
```
4
```

Indeed, it's simple to see that (check this decimal-to-binary converter)

$$10 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 8 + 0 + 2 + 0$$

therefore the 4 binary numbers (i.e. 0/1 bits) `1010` are sufficient to represent the integer number 10.

Python is very efficient in its internal representation of integer numbers as it can represent integers arbitrarily big, like 10^{100} (named Googol)

[3]:
```
googol = 10**100
print(googol)
```

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
```

[4]:
```
googol.bit_length()
```

[4]:
```
333
```

2 Floats

Non-integers numbers are represented in Python as `float` data type.

[5]:
```
q = 1/4
print(q)
```

```
0.25
```

[6]:
```
type(q)
```

[6]: `float`

The fraction $\frac{1}{4}$ is represented *exactly* as the `float` 0.25. This is because 0.25 has an exact (and obvious) binary representation (in terms of negative powers of the base 2)

$$\frac{1}{4} = (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) = (0 \times 1) + \left(0 \times \frac{1}{2}\right) + \left(1 \times \frac{1}{4}\right) = 0.25$$

where the 0/1 bits associated to smaller powers of 2: $ 2^{-3}, 2^{-4}, ...$ are all zero.

Therefore, in a *fixed-point* binary representation (that is a binary representation using a fixed number of bits after the decimal point '.', as the one above), the decimal number 0.25 can be represented as the binary number `0.01` (check this decimal-to-binary converter), that is using only the first two left-most bits after the '.' (which are the most significant).

Binary representation of `float` numbers is not always *perfect*. That is, it's not alway true that a decimal number $0 < q < 1$ can be represented exactly as the series

$$q = \sum_{i=1}^{k} b_i \times 2^{-i}$$

where $b_i = 0/1$ is the i-th bit. In particular it can be that:

- the series is infinite ($k = \infty$);
- the series requires more bits than those at disposal. That is, given a finite number of bits at disposal - say k_{MAX} - it can be that $k > k_{MAX}$.

In this last case, the best we can do is a *truncation* of the series. That is, q can will be approximately represented as

$$q \approx \sum_{i=1}^{k_{MAX}} b_i \times 2^{-i}$$

In real life things are more complicated. In particular, The IEEE 754 double-precision standard - currently adopted by modern 64-bits machines - reserves 64 bits to represent a decimal number, but bits are not simply associated to negative and decreasing powers of the base 2: $2^{-1}, 2^{-2}, \ldots$ as in the fixed-point binary representation that we considered before. The IEEE 754 standard prescribes a *floating-point* format, where the meaning and role of the bits in the binary representation changes depending on their position. In particular, for your knowledge (more informations in Wikipedia): - 1 bit (the 1st one) represents the *sign;* - 11 bits (from the 2nd to the 12th) represent an *exponent;* - 52 bits (from the 13th to the last one) represet the *fractional part.*

This representation allows to represent a greater range of decimal numbers, given the same amount of bits at disposal (64). This increase in the range of number representable comes at the cost of precision. In the IEEE 754 double-precision standards, the relative accuracy is of 15-digits.

The *finite-precision* in the binary representation of decimal numbers leads to expected results like:

[7]:
```
q = 0.25 + 0.1
q
```

[7]:
```
0.35
```

but also to unexpected ones like:

[8]:
```
q = 0.35 + 0.1  #should be 0.45
q
```

[8]:
```
0.44999999999999996
```

Nevertheless, module `decimal` allows us to set an arbitrary precision (we won't use it, but it's good for you to know):

[9]:
```
import decimal
from decimal import Decimal
```

[10]:
```
decimal.getcontext()
```

[10]:
```
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero,
Overflow])
```

the precision is of 28 significant (non-zero) digits by default (`prec=28`)

[11]:
```
q = Decimal(1) / Decimal(17)
q
```

[11]:
```
Decimal('0.05882352941176470588235294118')
```

the precision can be changed arbitrarily to set the number of significant (non-zero) digits after the '.'

[12]:
```
decimal.getcontext().prec = 3  # here we se the precision to 3 significant␣
↪digits after the '.'

q = Decimal(1) / Decimal(17)
q
```

[12]: Decimal('0.0588')

3 Bool

Logical states like `True` and `False` are represented in Python as `bool` data type.

The output of a *comparison* operator

- `<` (smaller than),
- `>` (greater than),
- `<=` (smaller or equal than),
- `>=` (greater or equal than),
- `==` (equal to),
- `!=` (not equal to)

is a boolean value.

[13]:
```
a = 17
b = -1
```

[14]:
```
flag = (a < b)
flag
```

[14]: False

[15]:
```
flag = (a > b)
flag
```

[15]: True

[16]:
```
flag = (a <= b)
flag
```

[16]: False

[17]:
```
flag = (a >= b)
flag
```

[17]: True

[18]:
```
flag = (a == b)
flag
```

[18]: False

[19]:
```
flag = (a != b)
flag
```

[19]: True

Logical operators - `and` (logic and), - `or` (logic or), - `not` (logic not)

apply to boolean values and return boolean values as output.

[20]:
```
bool_1 = (a < b)
bool_2 = (a > b)

print(bool_1)
print(bool_2)
```

```
False
True
```

[21]:
```
flag = (bool_1 and bool_2)
flag
```

[21]: False

[22]:
```
flag = (bool_1 or bool_2)
flag
```

[22]: True

[23]:
```
flag = (not bool_1)
flag
```

[23]: True

There is a one-to-one correspondence between `bool` and `int`:

[24]:
```
int(True)
```

[24]: 1

[25]:
```
bool(1)
```

[25]: True

where we have used the *casting* functions `int()` and `bool()` which cast a variable to `int` and `bool`, respectively (there are many more, `float()`, `str()`,…). More examples:

[26]:
```
i = int(1.15)
print(i)
type(i)
```

```
1
```

[26]: int

[27]:
```python
f = float(10)
print(f)
type(f)
```

```
10.0
```

[27]: float

[28]:
```python
s = str(90) # strings data types introduced below
print(s)
type(s)
```

```
90
```

[28]: str

Boolean values are particular useful within control flows structures. We examine here `while` loops and `if` conditions.

3.1 while loop

A `while` loop in Python is declared as follows:

```python
while condition:
    statement(s)
```

The Python interpreter evaluates the logical `condition`; if it is `True`, `statement(s)` (all the lines of code *indented* w.r.t. the `while` keyword) are executed. Then `condition` is evaluated again and, if `True`, the `statement(s)` are executed again. The loop ends when (and if) `condition` becomes `False`.

Warning: if `condition` never becomes `False`, you end up with an infinite loop, which will execute forever. Needless to say you should avoid infinite loops!

An academic example:

[29]:
```python
# while 1:
#    print("I'm into an infinite loop...")
```

To stop execution: - in a Jupyter Notebook: press *Kernel* and then *Interrupt* (see picture):

- in a script (using Spyder IDE): press the red square ( ) on the top-right angle of the *IPython console,* which is the interactive console on the bottom-right panel where you see the output of the code you type in the *Editor* (see picture). A `KeyboardInterrupt` message will confirm the stop.

Example: A Fibonacci number F_n is the sum of its two preceeding numbers. It is defined by the initial conditions:

$$F_0 = 0 F_1 = 1$$

and satisfies the relation

$$F_n = F_{n-1} + F_{n-2}$$

This code prints the first few Fibonacci numbers smaller than a constant C.

To do this we define a function `fib(C)` which takes in input the stopping constant `C`, uses a `while` loop to compute the numbers and returns the greatest number Fibonacci number satisfying condition $F_n < C$.

[30]:
```
def fib(C):
    """
    This function computes the greatest Fibonacci number smaller than consant␣
 ↪'C'.

    Parameters:
        C (float): stopping constant.

    Returns:
        F_n2 (int): greatest Fibonacci number smaller than C.
    """

    # initialization
    F_n2 = 0 # F_{n-2}
    F_n1 = 1 # F_{n-1}

    while F_n1 < C:

        # uncomment this line below if you want to print to screen the current␣
 ↪number
        # print(F_n2)

        # store F_{n-1} + F_{n-2} into a temporary variable
        temp = F_n1 + F_n2

        # update the last two numbers
        F_n2 = F_n1
        F_n1 = temp

    return F_n2
```

Let's set the stopping constant $C = 10^4$

[31]:
```
C_stop_const = 10000
```

[32]:
```
fib(C_stop_const)
```

[32]: 6765

You can print on screen the whole Fibonacci series until $F_n < C$ simly removing the comment at `print(F_n2)`.

We can easily test the speed of our code using the %timeit directive, which runs the same line of code (in this case the calls of function `fib(C)`) several times to have statistically significant averages of its running-time

[33]:
```
%timeit fib(C_stop_const)
```

```
1.04 µs ± 29 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Running `fib(10000)` takes on average $1.1 \mu s$.

We can exploit Python inline assignments to make this code more this function more compact.

[34]:
```
def fib_inline(C):
    """
    This function computes the greatest Fibonacci number smaller than consant␣
↪'C' using inline assignments.

    Parameters:
        C (float): stopping constant.

    Returns:
        F_n2 (int): greatest Fibonacci number smaller than C.
    """

    # inline initialization
    F_n2, F_n1 = 0, 1 # F_{n-2}, F_{n-1}

    while F_n1 < C:

        # uncomment this line below if you want to print to screen the current␣
↪number
        # print(F_n2)

        # inline update of the last two numbers
        F_n2, F_n1 = F_n1, F_n1 + F_n2

    return F_n2
```

[35]:
```
fib_inline(C_stop_const)
```

[35]: 6765

[36]:
```
%timeit fib_inline(C_stop_const)
```

```
1.09 µs ± 71.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

The inline assignments lead to a slightly performance improvement too.

3.2 if statement

An `if` statement in Python is declared as follows:

```
if condition:
    statement(s)
elif alternative_condition:
    statement(s)
else:
    statement(s)
```

The Python interpreter evaluates the logical `condition` next to the `if`; if it is `True`, the `statement(s)` indented under the `if` statement are executed. If `condition` is `False`, the logical `alternative_condition` next to the `elif` statement is evaluated; if it is `True` the `statement(s)` indented under the `elif` are executed. If also `alternative_condition` is `False`, the `statement(s)` indented under the `else` statement are executed.

Notice that: - `elif` statement is optional and there can be more than one; - `else` statement is optional.

[37]:
```
a = 17
b = -1

if a < b:
    print("a is smaller than b")
elif a == b:
    print("a equals b")
else:
    print("a is greater than b")
```

```
a is greater than b
```

4 Strings

One or more text characters are represented in Python as `str` data type. Use the double quotes `""` to define a string.

[38]:
```
s = "My name is Gabriele"
```

Type `str` is a very versatile data type. It can be splitted in single words (that is, considering as separator the white space):

[39]:
```
s.split()
```

[39]: ['My', 'name', 'is', 'Gabriele']

Moreover, strings can be *indexed* treating its single characters as elements of the strings:

[40]:
```
s = "IT_For_Business_And_Finance_2019_20"

print(s[0])
print(s[2])
```

```
I
_
```

and index access can be also from last character. Index `-1` points to the last one, `-2` to the second-last...

[41]:
```
print(s[-1])
print(s[-2])
```

```
0
2
```

Strings can be *sliced*. That is, you can select portions of it:

[42]:
```
s_slice = s[0:2] # characters from position 0 (included) to 2 (excluded)

print(s_slice)
type(s_slice)
```

```
IT
```

[42]: str

[43]:
```
s[2:5] # characters from position 2 (included) to 5 (excluded)
```

[43]: '_Fo'

[44]:
```
s[:2]   # character from the beginning to position 2 (excluded) --- equivalent
 ↪to s[0:2]
```

[44]: 'IT'

[45]:
```
s[-2:]  # characters from the second-last (included) to the end
```

[45]: '20'

Strings are *immutable*. If you try to change one of its characters, you get

```
TypeError: 'str' object does not support item assignment
```

[46]:
```
# s[0] = "A"
```

if you really want to modify a string, simply define a new one (even with the same name). Suppose you want to change the first element of string `s` from "I" to "A". You can do this way (notice the behavior of the `+` operator on strings):

[47]:
```
s = "A" + s[1:]
s
```

[47]:
```
'AT_For_Business_And_Finance_2019_20'
```

As all sequence-like data structures, strings have a length `len()`, which is the number of its characters.

[48]:
```
len(s)
```

[48]:
```
35
```

To conclude, a string in Python is a *data type* in that even a single character is a string

[49]:
```
type("c")
```

[49]:
```
str
```

but, as we have just seen, it also behaves as a sequence-like *data structure,* since it can be indexed, sliced, … as if it is an *array* of characters.