Python Programming Language

Assoc. Prof. Krzysztof Małecki

kmalecki.zut.edu.pl
p. 308 (WI2)

Python: from, how and who?

- Python is not a reptile - it's Monty Python from the Flying Circus
- Guido van Rossum (born 1960, Dutch) - created Python in December 1989, as he claims, actually out of boredom
- Guido joined Google in 2005 and moved to Dropbox in 2013.

Python: how it look like?

Python: what is he like?

- easy to learn and use, intuitive, with a high power of expression
- published as open source
- for general use, but also useful in dedicated applications
- shortens programming time and increases the comfort of the programmer's work

Python: has more than One Name

- two independent and incompatible versions of Python are currently in use:
  - Python 2 - kept alive due to the huge number of programs written in it (current version is 2.7)
  - Python 3 - used for new projects and treated as future-proof (current version is 3.12)

Python: what does it give us?

- the interpreter
- libraries of ready-made solutions that we can use in our code
- a simple developer environment called IDLE
- that is all we need to start our adventure with Python

Python

- two ways of work:
  - **interactive** - we type commands and Python executes them immediately
  - **non-interactive** - we write the source code in a file and then tell Python to execute that file

Python – syntax

- one statement on one line of the file
- the instruction does not end in any special way, you can insert empty lines into the code, if it improves the readability of the code
- a statement can be broken by putting a \ (backslash) at the end of a line and continuing on the next line

Example

```python
print("Hello")
print("My name is Python")
```

```python
print("Hello")

print \
(\
"My name is Python"\
)
```

Important

- you must not put whitespace at the beginning of a line if you don't know what for you are doing it!
- a line indented from the left margin is different to Python than the line that starts with the first column
- Python is case-sensitive: `X` for him is completely different from `x`
- consequently, you must write the function names exactly as it was written in the documentation, so:
  - this is OK → `print`
  - this is not OK → `Print` `PRINT` `PrInT`

A comment

- a comment begins where the # (hash) sign stands on a line and ends where that line ends
- remember: the hash inside the quotes doesn't start the comment
- block comment - to select a larger area (3 apostrophes at the beginning and at the end):

'''..........'''

A literal

- a **literal** is a data that signifies itself

- It is a literal → `3.1415926535`
- It is not a literal → π

Integer literals

- written as a string of Arabic numerals, without any inclusions (e.g. spaces)
  - OK → `3000000000`
  - NOT OK → `3 000 000 000`
- we can precede the number with the sign: `-` or the sign: `+`
- the following literals, despite some concerns, are valid numbers:

```
  -----123
+-+-+-+123
```

Integer literals

- if an integer literal begins with the prefix 0o (zero o), it means it was written in octal, for example:

```
0o20
```

and it is a value... $16_{(10)}$

- if an integer literal begins with the prefix 0x (zero x), it means that it was written in hexadecimal, for example:

```
0x20
```

and it is a value... $32_{(10)}$

Integer literals

- if an integer literal begins with the prefix 0b (zero b), it means it was written in binary, for example:

```
0b20
```

and it is a value... $??_{(10)}$

- ...but this example is OK:

```
0b110
```

and it is a value... $6_{(10)}$

Attention

- letters o, x and b can be uppercase:

```
0xFF      0XFF
0b0110    0B0110
0o777     0O777
```

- although the last one looks quite risky (note how much it depends on the font used - that's why there are special fonts for developers - e.g. Monaco is used in this presentation)

Real (floating point) literals:

- instead of the decimal comma (in Poland), we definitely use a dot!

```
2.49
```

- if there would only be a zero before the dot, it can be omitted (both literals below mean the same number):

```
0.49
.49
```

Real (floating point) literals:

- if there would only be a zero after the dot, it can be omitted (both literals below mean the same number):

```
49.0
49.
```

- but ... omitting the dot - although it seems to be meaningless - changes the character of the literal and therefore requires a moment of reflection:

`49` → it is an integer literal

`49.` → it is a real literal

Important

- the difference in the type of literal results in a different way of representing the number in computer memory, and hence in a completely different arithmetic used by the computer

Real (floating point) literals:

- very large and very small (as to the absolute value of a number) - real numbers can be written in the so-called scientific notation, e.g.

  `2.89E10` → $2.89 \cdot 10^{10}$

  `0.342E-20` → $0.342 \cdot 10^{-20}$

Logical literals

- they are used to write two elementary logical values, that is, true and false

```
True
False
```

- these literals should be written literally, for example, not like:

`TRUE` or `false`

Arithmetic literals

```
3 + 7.5   →   10.5
3 – 7.5   →   -4.5
3 * 7.5   →   22.5
3 / 7.5   →   0.4
10 // 4   →   2      //integer division
10 % 3    →   1      //modulo division (the rest of the division)
2 ** 8    →   256    // exponentiation
```

Operator priorities (from highest to lowest)

```
1. **
2. *  /  %   //
3. +  -
```

but:

- the natural order of calculations can be changed using brackets
- we use only normal brackets (and)
- pairs of brackets can be freely nested, for example:

```
(((((((((2 + 2)))))))))
```

- specialized editors (e.g. IDLE, PyCharm) will help us with this ☺

Bitwise operators

- the operators **described earlier** work equally well with real and integers

- besides them, there are only integer operators

- these are the so-called **bitwise operators**, because they affect each bit of data separately

bin() function

- useful for experiments with bitwise operators

```
bin(x):
```

- an argument: integer
- the result: the binary form of the argument

```
print(bin(15)) → 0b1111
```

hex() function

hex(x):

- an argument: integer
- the result: the hexadecimal form of the argument

print(hex(15)) → 0xf

Bitwise operators

`~x` → negation `print(bin(~0b011))` → `-0b100`

`x & y` → bit product `print(bin(0b110 & 0b011))` → `0b10`

`x | y` → bit sum `print(bin(0b110 | 0b011))` → `0b111`

`x ^ y` → xor `print(bin(0b110 ^ 0b011))` → `0b101`

`x << y` → left shift `print(bin(0b110 << 2))` → `0b11000`

`print(0b110 << 2)` → `24`

`x >> y` → right shift `print(bin(0b110 >> 1))` → `0b11`

`print(0b110 >> 1)` → `3`

Relational operators

```
x > y
x >= y
x < y
x <= y
x == y
x != y
```

the result of these operators is always `True` or `False`

Abbreviated operators

- the convention is as follows:

```
A = A OP B        →    A OP= B
```

```
A = A + B    →   A += B
A = A - B    →   A -= B
A = A * B    →   A *= B
A = A / B    →   A /= B
A = A // B   →   A //= B
A = A ** B   →   A **= B
```

Abbreviated operators

```
A = A << B   →   A <<= B
A = A >> B   →   A >>= B
A = A & B    →   A &= B
A = A | B    →   A |= B
A = A ^ B    →   A ^= B
```

Variables – rules of name

- can contain letters (upper and lower case), numbers and the character _ (underscore)
- may contain national characters
- uppercase and lowercase letters are treated as different
- cannot start with a digit
- the use of certain variable must be preceded by giving the value for this variable

The operator = (assignment operator)

- the effect: assigning the value of the expression being on the right side of the operator to the variable listed on the left
- the result: the value of the expression being on the right side of =

- it means that the assignment:

```
a = b = c = d = 1
```

  should be understood as a sequence of assignments:

```
a = (b = (c = (d = 1)))
```

- and consequently:

```
a = 1
b = 1
c = 1
d = 1
```

Remember

- Python likes short forms of expressions:

```
a = 1
b = 2

a,b = b,a
```

```
>>> a=1
>>> b=2
>>> print(a,b)
1 2
>>> a,b=b,a
>>> print(a,b)
2 1
```

input() - variant #1

- we will use it to take data from the user

```
input()
```

- an argument: none
- the effect: loading **a line of data** from the console
- the result: user-entered string
- e.g.:

```
text = input()
```

input() - variant #2

- we will use it to take data from the user

```
input(x)
```

- an argument: hint for the user
- the effect: loading **a line of data** from the console
- the result: user-entered string
- e.g.:

```
text = input("Give a string: ")
```

Remember

- input() function always loads text (string)
- string is not a number (even if it consists of digits)
- if you want to use the entered text as a number, you have to convert explicitly (transforming a string to the internal representation of a number)

Example

```
>>> x=input()
123
>>> y=x/3
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    y=x/3
TypeError: unsupported operand type(s) for
/: 'str' and 'int'
>>> y=int(x)/3
>>> print(y)
41.0
```

int()

converts a string to an integer

```
int(x)
```

an argument: a string representing the number

the effect: a string conversion to an integer

the result: a converted number

attention: gives an exception on failure

e.g.:

```
number = int(input())
```

float()

converts a string to a float

```
float(x)
```

an argument: a string representing the number

the effect: a string conversion to a float

the result: a converted number

attention: gives an exception on failure

e.g.:

```
price = float(input())
```

Remember

- the int() and float() functions trust that the argument passed to them is really a notation of a number
- otherwise the functions will be disappointed...

```
>>> x=int(input())
bulbulator
>>> y=x/3
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    x=int(input())
ValueError: invalid literal for int() with base 10:
'bulbulator'
```

Example

- a program that squares a number:

```
number=float(input("Enter a number, please: "))
square=number ** 2
print("Square of ", number, " is ", square)
```

A module

- a module is a code that does not run directly, and you use the facilities it contains (e.g. functions)
- to use a certain facility, it **must be imported** from the module

`math`

- this module contains a number of mathematical functions

`sqrt`

- a certain function from `math` module, computing the square root

Import – variant #1

```
import math
```

the effect:

- all the facilities of the `math` module become available, but...
- they should be identified with the so-called **qualified name**, e.g .:

```
y = math.sqrt(x)
```

Import – variant #2

```
from math import sqrt
```

the effect:

- only the explicitly mentioned facilities from the `math` module become available, but …
- they do not need to be identified by a **qualified name**

```
y = sqrt(x)
```

Can we take a negative number to the root?

```
Traceback (most recent call last):
  File "prog.py", line 4, in <module>
    y=sqrt(number)
ValueError: math domain error
```

We need to branch the code

- if the value is non-negative, we will take the square root

- otherwise we will do nothing

if – variant #1

```
if expression:
      a part of code
```

expression:

- logical (boolean) expression
- if it is `True`, the `if` instruction determines that some statements **must be executed**
- otherwise a part of code will be omitted

Remember

- statements being the content of an `if` instruction is indicated by the indentation level in Python! (relative to the left margin)
- indentation can be obtained with spaces or tabs
- the second option is recommended
- mixing both variants is **risky**
- returning to the previous indent level marks the end of the `if`

Example

```
from math import sqrt

number=float(input("Enter a number, please: "))
if number >= 0.0:
   s=sqrt(number)
   print(s)
```

if – variant #2

```
if expression:
      a part of code1
else:
      a part of code2
```

expression:

- if expression is `True`, the `if` instruction consider that *a part of code1* **must be executed**
- otherwise *a part of code2* will be executed

if – variant #3

```
if expression1:
        a part of code1
elif expression2:
        a part of code2
else:
        a part of code3
```

expression:

- if expression1 is `True`, the `if` instruction consider that *a part of code1* **must be executed**
- if expression1 is `False`, expression2 will be checked and if it is `True`, *a part of code2* will be executed
- otherwise *a part of code3* will be executed

Example

```
from math import sqrt

number=float(input("Enter a number, please: "))
if number == 0.0:
   print("It is known without computing..., zero!")
elif number > 0.0:
   s=sqrt(number)
   print(s)
else:
   print("You entered incorrect data!")
print("The end")
```

if – some remarks

- the phrase `elif` can appear multiple times, but only after an `if`
- the phrase `elif` may also not appear at all
- the `else` phrase may occur **only once** and must be **the last one**
- none of these phrases can occur without the previous `if` !!!

while – variant #1

```
while expression1:
    code1
```

- as long as *expression1* equals `True`, the while statement will execute *code1*
- if *expression1* is `False`, *code1* will be skipped

while – variant #2

```
while expression1:
        code1
else:
        code2
```

- as long as *expression1* equals `True`, the while statement will execute *code1*
- if *expression1* is `False`, *code2* will be executed (**at least once**)

Remember

- if one if / while / ... statement is contained in another if / while / ... statement, it manifests itself with increasing indentation
- be careful when you use indentation - wrong indents will result in bad code behaviour
- errors of this kind are **difficult to find** ☹

a `sleep()` function from `time` module

suspending the program for the number of seconds indicated

```
sleep(n)
```

an argument: the number of seconds

the effect: waiting for the indicated number of seconds

- e.g.:

`time.sleep(3600)` ← wait one hour

Example1 of `while`

```python
from time import sleep

timer=int(input("Enter a number of seconds: "))
while timer > 0:
    print(timer, "...")
    timer = timer - 1
    sleep(1)
print("The countdown is complete. Boom!")
```

Example2 of `while` (a better ver.)

```python
from time import sleep

timer = 0
while timer <= 0:
    timer=int(input("Enter a number of seconds: "))
    if timer <= 0:
        print("Enter a non-negative value!")
while timer > 0:
    print(timer, "...")
    timer = timer - 1
    sleep(1)
print("The countdown is complete. Boom!")
```

print() – a little explanation

A set of `print()` statements as:

```
print("Cat")
print("and")
print("dog")
```

will display on the screen:

```
Cat
and
dog
```

print() – a little explanation

...but a set of `print()` statements as:

```
print("Cat", end=" ")
print("and", end=" ")
print("dog")
```

will display on the screen:

```
Cat and dog
```

print() – a little explanation

...but a set of `print()` statements as:

```
print("Cat", end="")
print("and", end="")
print("dog")
```

will display on the screen:

```
Catanddog
```

for – variant #1

```
for x in range(min,max):
    part of code
```

the so-called **control variable** - assumes successive values in subsequent loops; it still has the last used value after the loop is finished

function that creates a range (a list) with extremes defined by parameters

Attention!!!

`range(x,y)`

generates a list of values:

x, x+1, x+2, ... , y-2, y-1

`range(0,max)`

can be shortened:

`range(max)`

for – variant #2

```
for x in range(min,max):
    code1
else:
    code2
```

- *code2* will be executed when the values after the `in` phrase are finished

for – variant #2

```
for number in range(0,5):
      print(number, end=" ")
else:
      print("!")
```

```
0 1 2 3 4 !
```

for

```
for x in reversed(range(min,max)):
```

function that inverts the obtained range (list)

```
for number in reversed(range(0,5)):
     print(number, end=" ")
```

```
4 3 2 1 0
```

Statements that control the execution of a loop

- if you want to exit the loop earlier than the loop timer indicates

```
break
```

- if you want to start the next iteration of the loop earlier

```
continue
```

and finally a riddle - what this program does?

```
for w in range(3):
    x = 20
    s = 1
    for l in range(5):
        for spaces in range(x):
            print(end=" ")
        for stars in range(s):
            print("*", end="")
        print()
        x = x - 1
        s = s + 2
```

☺

```
    *
   ***
  *****
 *******
*********
    *
   ***
  *****
 *******
*********
    *
   ***
  *****
 *******
*********
```

Thank you for your attention

see you at the next lecture