# Programming with Python

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

## Contents:

1. Ten commandments of programming
2. Loops
3. Comparison and logical operators
4. Conditionals

## Ten commandments of programming

1. First think about **what exactly you want to do** before writing any code.
2. Expectation: Break your problem into **small independent blocks.** Specify precisely what you want each block to do.
3. Prepare the project, i.e., create a folder and start with a **fresh .py file.**
4. Work "Inside to Outside": Identify the "core functionality" in each block, **make the core work first,** and only then refine and extend.
5. If necessary, **use the internet** to get help (google "Python how to ...").
6. However, do not just copy-and-paste. Understand the main idea and **write your own code!**
7. Verification: Test your code **line by line** as you write it! Your computer always does exactly what you tell it to do. So if the result is different from the expectation defined in Step 2, it's your fault.
8. **Read Python error messages.** If you just added one line and the code "doesn't work" anymore, the problem is likely related to this line.
9. Always put **meaningful comments** into your code as you write it. Undocumented code is useless!
10. Be proud of what you achieved. Create a Github profile and share your codes.

# Loops

It is rarely useful that a computer performs each operation only once; we usually need it to repeat what it does. To do so, we need *loops*.

## `for`-loops

The most common type is a `for`-loop, which is usually used to execute some part of the code a predetermined number of times.

In Python 3, we often use the `for`-loop together with the `range` function which *pretends* to return a list of numbers (it returns something more complex, we can consider it a list for now). That function can be called as follows:

- `range(n)` -- numbers $0, 1, \ldots, n-1$; this is equivalent to `range(0,n)`;
- `range(m, n)` -- numbers $m, m+1, \ldots, n-1$;
- `range(m, n, s)` -- numbers $m, m+s, \ldots, m+sk$, where $k \in \mathbb{N}$ such that $m + sk < n \leq m + s(k+1)$.
  In other words, numbers from $m$ to $n-1$ with step $s$, but we might not hit $n-1$, depending on the value of step $s$.

Do not forget that the ending is **not included**, hence "$-1$"!

**Python 2 remark:** In Python 2, `range` actually returns a list. The exact equivalent to `range` from Python 3 is called `xrange` in Python 2.

## `for`-loop in action

Let us make an overly enthusiastic version of the "Hello, World!" program that would repeat its message $5$ times.

In [1]:

```
for i in range(5):
    print("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

## How does this work?

Let us read it as if it was a sentence in regular English language, taking into account that `range(5)` acts as a list of numbers $0, 1, 2, 3, 4$

> For `i` in [0, 1, 2, 3, 4], print "Hello, World!".

So, this is equivalent to the code:

In [2]:

```
  = 0
 rint("Hello, World!")
  = 1
 rint("Hello, World!")
  = 2
 rint("Hello, World!")
  = 3
 rint("Hello, World!")
  = 4
 rint("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Obviously, we don't need the variable `i`, but it's part of the loop. If we want to ignore it, we can use the underscore _ instead:

In [3]:

```
for _ in range(5):
    print("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

This is the same as

In [4]:

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

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

## `for`-loop syntax

`for`-loops have the following elements:

1. start with the keyword "`for`",
2. followed by the name of the variable that will be assigned all the values through which we want to loop (or "_" if we don't need those values),
3. then the keyword "`in`",
4. then a list or something that acts like it (we'll see more examples throughout the course), and
5. then a colon "`:`".

**Do not forget the colon!** This will make Python somewhat nervous...

## Indentation ← VERY IMPORTANT!

Notice the indentation in the second line:

In [5]:

```
for i in range(5):
    print("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

All the commands with the same indentation "belong" to that loop. Observe the difference:

In [6]:

```
rint("The first loop:")

or i in range(3):
   print("Inside loop.")
   print("And where is this? Inside or outside of the loop?")

rint()
rint("The second loop:")

or i in range(3):
   print("Inside loop.")
rint("And where is this? Inside or outside of the loop?")
```

```
he first loop:
nside loop.
nd where is this? Inside or outside of the loop?
nside loop.
nd where is this? Inside or outside of the loop?
nside loop.
nd where is this? Inside or outside of the loop?

he second loop:
nside loop.
nside loop.
nside loop.
nd where is this? Inside or outside of the loop?
```

Blocks of commands, like the ones above, are defined via indentation, **and indentation alone!** There are no curly brackets `{...}` like in C-like languages, no `begin...end` like in Pascal, no `end` like in MATLAB,... **only indentation, so pay close attention to it!**

Empty lines are completely ignored. Do use them for better code readability!

## A more useful example

Write a program that inputs an integer $n$, then inputs $n$ numbers $a_1, a_2, \ldots, a_n$ and prints

$$S = \sum_{k=1}^{n} a_k.$$

How would we do it manually?

1. Input $n$.
2. Initialize $S$ to $0$ (meaning give $S$ its initial value $0$).
3. Then input a number, say $x$ (by "number", we usually mean a "real number", but it can be an integer or something else, depending on the context).
4. Add $x$ to $S$, i.e., $S \leftarrow S + x$.
5. Repeat steps 3 and 4 a grand total of $n$ times.
6. Print the result $S$.

Since computers prefer to know in advance that we wish to loop through some of the code, we shall rearrange the above (i.e., put the item 5 where the loop starts):

1. Input $n$.
2. Set $S = 0$.
3. Repeat the following steps $n$ times
   A. Input a number $x$.
   B. Add $x$ to $S$, i.e., $S \leftarrow S + x$.
4. Print the result $S$.

We are now ready for a Python code:

In [ ]:

```
  = int(input("How many numbers do you want to add up? "))
  = 0
or i in range(n):
   x = float(input("Input a_" + str(i+1) + ": "))
   s += x
rint(s)
```

In the above code, we use the variable `i` to explain to the user which number they have to input. However, since `range(n)` traverses through numbers $0, 1, \ldots, n-1$ and we want then to be $1, 2, \ldots, n$, we used `i+1` for such a description.

We can avoid this extra addition by simply explaining to the `range()` function that we want the numbers to start from $1$ and go up to $n$ (i.e., numbers from $1$, strictly smaller than $n+1$):

In [ ]:

```
  = int(input("How many numbers do you want to add up? "))
  = 0
or i in range(1, n+1):
   x = float(input("Input a_" + str(i) + ": "))
   s += x
rint(s)
```

Usually, the first approach is better because indexing of lists (we'll get to them later) is zero-based and you'll almost always want your loop counter to respect that.

## `while`-loops

It is not always known ahead of the execution how many times the loop will have to repeat its block. For example:

- Load some numbers until a zero is loaded and do something with them;
- Do something with number's digits (technically, we can count the digits first, but that'd be a waste of time);
- Read the data as long as there is input (from a file, standard input, etc.).
  ...

Equivalent to the basic `for`-loop,

In [9]:

```
  = int(input("n = "))
 or i in range(n):
    print(i)
```

```
n = 5
0
1
2
3
4
```

is the following code:

In [10]:

```
  = int(input("n = "))
  = 0
 hile i < n:
    print(i)
    i += 1
```

```
n = 5
0
1
2
3
4
```

## How does this work?

Let us, as before, read the above code as if it was a sentence in regular English language, taking into account that `i += 1` stands for "add 1 to `i`", i.e., "increase `i` by 1":

> While $i$ is less than $n$, print $i$ and increase it by one.

And this is exactly how `while`-loops work:

1. check the condition,
2. if the condition is true, execute the loop's body and go back to step 1,
3. if the condition is false, skip the body and continue the execution behind the loop.

## `while`-loop syntax

`while`-loops have the following elements:

1. start with the keyword "`while`",
2. followed by the condition that has to be checked,
3. then a colon "`:`".

As with the `for`-loop, **do not forget the colon!**

## When is the condition checked?

Before each execution of the body, **and only then!**

Observe the following code:

In [5]:

```
 = 0
hile i < 2:
   print("Before the increment:", i)
   i += 1
   print("After the increment:", i)
```

```
efore the increment: 0
fter the increment: 1
efore the increment: 1
fter the increment: 2
```

So, even after `i` took the value 2 (thus, making the condition `i < 2` false), the body still executed until the end, before rechecking the condition and terminating the loop!

### Can we change how the loops run?

Yes, we can break the loop earlier than it would normally stop and we can skip parts of it. This will be addressed later on, along with other methods of changing the normal flow of control in programs.

# Comparison and logical operators

These are the comparison operators that can be used on numbers (and some other types, for example strings):

| Operator | Example | Meaning |
|---|---|---|
| `<` | `a < b` | The value of `a` is smaller than the value of `b` |
| `<=` | `a <= b` | The value of `a` is smaller than or equal to the value of `b` |
| `>` | `a > b` | The value of `a` is bigger than the value of `b` |
| `>=` | `a >= b` | The value of `a` is bigger than or equal to the value of `b` |
| `==` | `a == b` | The value of `a` is equal to the value of `b` (but not necessarily identical!) |
| `!=` | `a != b` | The value of `a` is not equal to the value of `b` |
| `is` | `a is b` | `a` and `b` are exactly the same object |
| `is not` | `a is not b` | `a` and `b` are not exactly the same object |
| `or` | `a or b` | `a` is true or `b` is true (or both); `a` and `b` can be non-Boolean values |
| `and` | `a and b` | Both `a` and `b` are true; `a` and `b` can be non-Boolean values |
| `not` | `not a` | `True` if `a` is false; `False` otherwise; `a` can be a non-Boolean value |

When checking if the value of a variable `x` is undefined (i.e., it is `None`), always use `x is None` or `x is not None`, and **not** `x == None` or `x != None` (the reasons are quite technical and far beyond the scope of this course; those interested can read more about the subject here (http://jaredgrubb.blogspot.co.uk/2009/04/python-is-none-vs-none.html)).

Also, do not use `not x is None` or `not (x is None)`; instead, use a much more readable `x is not None`.

# A note on composite comparisons

As we have seen, `a < b` is `True` if `a` is less than `b` and `False` otherwise:

In [12]:

```
rint("1 < 2:", 1 < 2)
rint("1 < 1:", 1 < 1)
rint("2 < 1:", 2 < 1)
```

```
1 < 2: True
1 < 1: False
2 < 1: False
```

In Python, we can also use

```
    < b < c
```

as a shorthand for

```
    < b and b < c
```

and

```
    < b > c
```

as a shorthand for

```
    < b and b > c
```

as shown below:

In [13]:

```
rint("1 < 2 < 3:", 1 < 2 < 3)
rint("1 < 2 < 1:", 1 < 2 < 1)
rint("1 < 2 > 3:", 1 < 2 > 3)
rint("1 < 2 > 1:", 1 < 2 > 1)
```

```
1 < 2 < 3: True
1 < 2 < 1: False
1 < 2 > 3: False
1 < 2 > 1: True
```

Of course, all of the comparison operators can be used (do try to not sacrifice the code readability):

In [14]:

```
  = 1
  = 2
  = 2
  = 0
 rint("1 < 2 == 2 >= 0:", a < b == c >= d)
 rint("1 < 2 == 0 >= 2:", a < b == d >= c)
 rint("1 < 2 >= 2 > 0: ", a < b >= c > d)
 rint("1 <= 2 >= 2 > 0:", a <= b >= c > d)
```

```
  < 2 == 2 >= 0: True
  < 2 == 0 >= 2: False
  < 2 >= 2 > 0:  True
  <= 2 >= 2 > 0: True
```

However, **be very careful in other languages**: the form with more than one comparison will be accepted by almost all of them, but it will not work as one would expect (like it does in Python)! This affects C/C++, MATLAB, PHP, PERL, Java, JavaScript,... It does work as expected in Mathematica. In R, it won't work at all (but at least it will report an error).

# A note on (in)equality operators

The equality == and inequality != operators ignore the types of data to a certain extent. For example,

In [15]:

```
 rint("False == 0:", False == 0)
 rint("  type(False):", type(False))
 rint("  type(0):", type(0))
 rint("  type(False) == type(0):", type(False) == type(0))
 rint("True == 1:", True == 1)
 rint("  type(True):", type(True))
 rint("  type(1):", type(1))
 rint("  type(True) == type(1):", type(True) == type(1))
 rint("True != 2:", True != 2)
 rint("  type(True):", type(True))
 rint("  type(2):", type(2))
 rint("  type(True) == type(2):", type(True) == type(2))
```

```
False == 0: True
  type(False): <class 'bool'>
  type(0): <class 'int'>
  type(False) == type(0): False
True == 1: True
  type(True): <class 'bool'>
  type(1): <class 'int'>
  type(True) == type(1): False
True != 2: True
  type(True): <class 'bool'>
  type(2): <class 'int'>
  type(True) == type(2): False
```

Some languages (like PHP and JavaScript) have a *strict equality operator* `===` and *strict inequality operator* `!==` that check both types and values of the compared objects. However, Python doesn't have that.

# Conditionals

We have seen how to say

> While some condition is true, do this.

However, we don't always want to repeat doing things. Sometimes, we want:

> If some condition is true, do this.

This request leads us to *conditionals*.

A conditional is created with `if` statement and it is very similar to `while`-loops, with the main difference being that its block will be executed at most once (if the condition is true).

Since the execution of the body is not repeated by the `if` statement, there are two possible branches:

1. the one that is executed **if the condition is true**,
2. the one that is executed **if the condition is false**.

However, sometimes, we want to say

> If `condition1` is true, do `job1`, else if `condition2` is true, do `job2`, else do `job3`

Here is an example of all three of these:

In [16]:

```
 = int(input("Input an integer: "))
f n < 0:
   print("Number", n, "is negative.")
lif n > 0:
   print("Number", n, "is positive.")
lse:
   print("Number", n, "has an identity crisis!")
```

```
Input an integer: 17
Number 17 is positive.
```

Notice how `else if` is abbreviated to `elif`. This is to avoid too much indentation. Without it, we'd have to write:

In [17]:

```
 = int(input("Input an integer: "))
f n < 0:
   print("Number", n, "is negative.")
lse:
   if n > 0:
       print("Number", n, "is positive.")
   else:
       print("Number", n, "is a rebel!")
```

```
Input an integer: 0
Number 0 is a rebel!
```

As before, all the controlling parts of a conditional (`if condition`, `elif condition`, and `else`) **need to end with a colon!**

Of course, we can stack as many `elif` parts as we need (including none of them), and we can omit the `else` part if we don't need it.

## Using `else` with loops

In Python 3, it is possible to use `else` in conjuction with loops. However, it is an unusual construct that other languages do not use and is best avoided by new programmers. At this point, we haven't covered enough of Python to explain what it does.

So be careful **not** to write

```
   hile condition1:
      if condition2:
          some_code
   lse:
      some_code
```

instead of

```
   hile condition1:
      if condition2:
          some_code
      else:
          some_code
```

as Python will not report an error. It will instead do *something*, but not what you wanted.

We shall cover loops' `else` statement later on in the course.

# An example: minimum and maximum

Give two variables `a` and `b`, find which one is the minimum and which one is the maximum. Store those values in variables `min_value` and `max_value`, respectively.

In [18]:

```
 We load some values, so we can test the code
 = float(input("a = "))
 = float(input("b = "))
 The solution to the problem
f a < b:
   min_value = a
   max_value = b
lse:
   min_value = b
   max_value = a
 Print the result, to check if it's correct
rint("a = ", a)
rint("b = ", b)
rint("Min:", min_value)
rint("Max:", max_value)
```

```
a = 19
b = 17
a =  19.0
b =  17.0
Min: 17.0
Max: 19.0
```

Since Python has functions `min` and `max`, we can do this in a more Pythonic way:

In [19]:

```
  We load some values, so we can test the code
  = float(input("a = "))
  = float(input("b = "))
  The solution to the problem
 in_value = min(a, b)
 ax_value = max(a, b)
  Print the result, to check if it's correct
 rint("a = ", a)
 rint("b = ", b)
 rint("Min:", min_value)
 rint("Max:", max_value)
```

```
  = 19
  = 17
  =  19.0
  =  17.0
 in: 17.0
 ax: 19.0
```

# (Sub)example

Load a and b in a way that a <= b.

This is wrong:

In [20]:

```
  = float(input("a = "))
  = float(input("b = "))
 f a > b:
    # a and b are in a wrong order, so we swap their values
    a = b
    b = a
  Print the result, to check if it's correct
 rint("Loaded:")
 rint("  a =", a)
 rint("  b =", b)
```

```
a = 19
b = 17
Loaded:
  a = 17.0
  b = 17.0
```

This is correct:

In [21]:

```
  = float(input("a = "))
  = float(input("b = "))
f a > b:
   # a and b are in a wrong order, so we swap their values
   tmp = a
   a = b
   b = tmp
 Print the result, to check if it's correct
rint("Loaded:")
rint("  a =", a)
rint("  b =", b)
```

```
a = 19
b = 17
Loaded:
  a = 17.0
  b = 19.0
```

A Pythonic solution would be:

In [22]:

```
  = float(input("a = "))
  = float(input("b = "))
f a > b:
   # a and b are in a wrong order, so we swap their values
   (a, b) = (b, a)
 Print the result, to check if it's correct
rint("Loaded:")
rint("  a =", a)
rint("  b =", b)
```

```
a = 19
b = 17
Loaded:
  a = 17.0
  b = 19.0
```

Please, **do not use this** until you know how and why it works.

# An example: digits of a number

Here is the problem:

> Find the sum of a given number's digits.

While there is a very "Pythonic" way to get the digits of a number, we shall do the usual integer arithmetic algorithm that is easy to port to all major modern languages.

How do we find digits of a number x?

The last one is easy: `x % 10` is the remainder of the division $x/10$. But what about the rest?

Here is a simple trick: once we're done with the last digit, we don't need it anymore and we can discard it. How do we do that?

Hint: we have just used the % operator to get the last digit. Is there an operator that gives x **without** the last digit?

Of course there is: operator `//` (integer division) is complementary to %. More precisely, `x // 10` gives us x without the last digit.

In [23]:

```
  = int(input())
rint("x =", x)
rint("x % 10 =", x % 10)
rint("x // 10 =", x // 10)
```

```
1719
x = 1719
x % 10 = 9
x // 10 = 171
```

A word of warning: try the above code with a negative x!

How do we deal with this?

There is a nice function for getting the absolute value of a number, conveniently called `abs()`:

In [24]:

```
  = abs(int(input()))
rint("x =", x)
rint("x % 10 =", x % 10)
rint("x // 10 =", x // 10)
```

```
-1719
x = 1719
x % 10 = 9
x // 10 = 171
```

So, here is a sketch of how to solve the above problem (sum of digits):

1. initialize `s = 0`,
2. load an integer and save its absolute value to `x`,
3. get the last digit from `x` and add it to `s`,
4. remove that last digit from `x`,
5. repeat the steps 3-4 until `x` has no more digits (i.e., it drops to zero).

We'll add additional `print()` to the loop, to follow the states of the variables.

In [25]:

```
 = 0
 = abs(int(input()))
rint("x =", x)
hile x > 0:
   s += x % 10
   x //= 10
   print("sum = " + str(s) + "; x =", x)
rint("final sum =", s)
rint("final x =", x)
```

```
-1719
x = 1719
sum = 9; x = 171
sum = 10; x = 17
sum = 17; x = 1
sum = 18; x = 0
final sum = 18
final x = 0
```

**Important:** Notice the final value of `x` in the last line -- it is zero. Doing the above, the value of the variable is **lost** and cannot be recreated!

If we need the value of `x` afterwards, we have to keep it in another variable (or do the whole thing in a function, as we shall soon see):

In [26]:

```
  = 0
  = abs(int(input()))
mp = x
rint("x = " + str(x) + "; s = " + str(s) + "; tmp = " + str(tmp))
hile tmp > 0:
   s += tmp % 10
   tmp //= 10
   print("x = " + str(x) + "; s = " + str(s) + "; tmp = " + str(tmp))
rint("Final values: x = " + str(x) + "; s = " + str(s) + "; tmp = " + str(tmp))
```

```
-1719
  = 1719; s = 0; tmp = 1719
  = 1719; s = 9; tmp = 171
  = 1719; s = 10; tmp = 17
  = 1719; s = 17; tmp = 1
  = 1719; s = 18; tmp = 0
inal values: x = 1719; s = 18; tmp = 0
```

By the way, there is a shorter way to do this, but specific to Python:

In [27]:

```
rint(sum(int(d) for d in str(abs(int(input())))))
```

```
-1719
18
```

We'll be able to understand the above code later on.

# A bit more on Boolean expressions

Whenever we write

```
f some_condition:
   do_something()
```

we are effectively saying

> If `some_condition` is true, then `do_something()`.

This is pretty straighforward when `some_condition` has a Boolean value. However, one can use most of the types in a condition. In this case, we say that we are using those values in a *Boolean context*.

Some general rules of evaluation in Boolean context are:

1. `None` and 0 (zero) evaluate to `False`.
2. Nonzero numbers evaluate to `True`.

Some examples:

In [28]:

```
 = 0
 = 1-1
f 0:
   print("0 is true.")
lse:
   print("0 is false.")
f 17:
   print("17 is true.")
lse:
   print("17 is false.")
f x:
   print("x is true.")
lse:
   print("x is false.")
f y:
   print("y is true.")
lse:
   print("y is false.")
f 1-1:
   print("1-1 is true.")
lse:
   print("1-1 is false.")
```

```
0 is false.
17 is true.
x is false.
y is false.
1-1 is false.
```

As we learn new data structures, we will mention their Boolean interpretations as well. However, a general rule is: if the value resembles a zero or if it is empty, its Boolean interpretation is `False`; otherwise, it is `True`. If unsure you can easily check for yourself:

In [29]:

```
rint(bool(1))
```

```
True
```