03 – Control structures

Bálint Aradi

Scientific Programming in Python (2026)

https://atticlectures.net/scipro/python-2026/

Branching

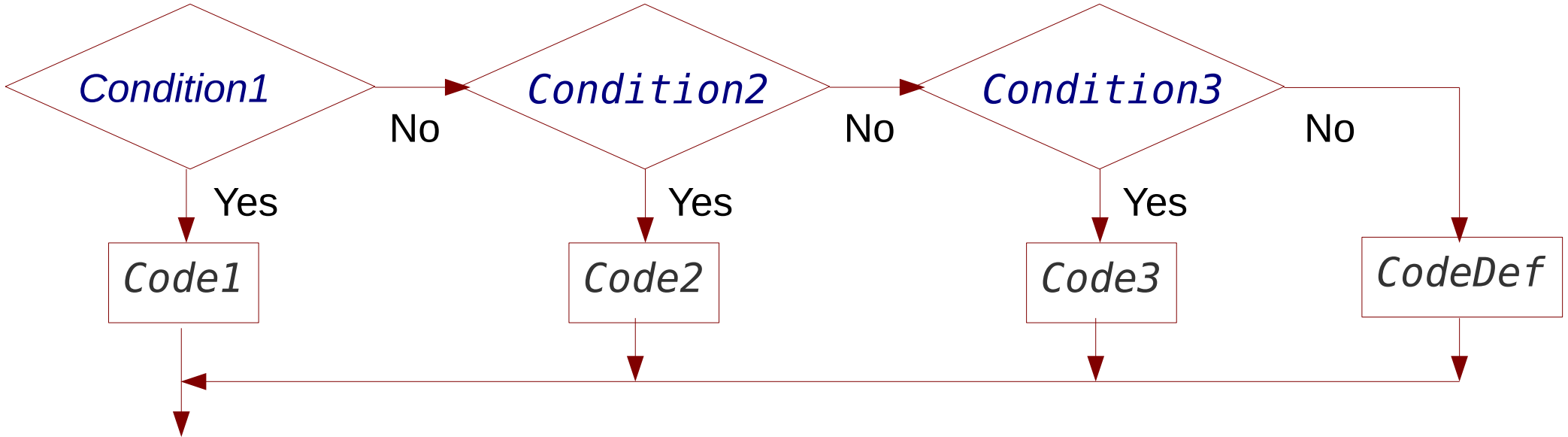
- Optional code execution based on condition evaluation

```
if Condition1:
    Code1
elif Condition2:
    Code2
elif Condition3:
    Code3
else:
    CodeDef
```

Indentation signalises nesting

Start of a nested block

- Nested blocks in Python **start with colon (:)**
- One should always use **4 spaces as indentation**
- End of nested block is signalised by an **unindented statement**



Indentation in Python

- Indentation is not optional, but part of the language semantics
- Indentation signalises nesting
- Amount of indentation signalises nesting depth
- Each nested block should be indented by exactly 4 space characters
- Inconsistent indentation leads either to syntax error or to wrong code logics

```
if answer[0] == "y":
    print("OK, you agree")
else:
    print("I see")
    print("You don't agree")
print("Let's continue")
```

Indented, belongs to if-block
(Only executed if answer[0] == “y”)

Indented, belongs to else-block
(Only executed if answer[0] != “y”)

Unindented, outside of if/else block
(Always executed)

- Use an editor which supports Python to ensure proper indentation!

If-else expression

- One can choose between two expressions with an if/else construct within an expression
- Use it only for trivial (short) cases

Syntax:

```
true_expression if condition else false_expression
```

```
mytype = "pos. semidef" if b >= 0 else "negative"
print("b is of type:", mytype)
```

Evaluation as bool expression

- Each object can be evaluated as a bool expression
- Evaluation is type dependent: Numerical types are usually False, if their value is zero. Container types are usually False, if they are empty

Object type	Evaluated to False	Evaluated to True
bool	False	True
int	0	any other value
float	0.0	any other value
complex	0.0+0.0j	any other value
string	"" (empty string)	contains at least one char.
list	[] (empty list)	contains at least one element
dict	{} (empty dict)	contains at least one element

```
if num % 2:
    print("odd")
```

`if num % 2 != 0`

```
if not num % 2:
    print("even")
```

`if num % 2 == 0`

for loop

- Iteration over given values can be realised with a **for-loop**

```
for loop_variable in iterable_object:
    loop code
```

Use the **for** loop, if the nr. of iterations is **known** in advance

- The iterable object can be anything, which is able to return values one-by-one (implements the iterator-interface)
- Example: string is iterable, it returns its characters one by one:

```
name1 = 'john'
for char in name1:
    print("Char: ", char)
```

```
Char:  j
Char:  o
Char:  h
Char:  n
```

- If loop variable is not needed, use _ as a placeholder:

```
for _ in range(4):
    tt.left(90)
    tt.forward(10)
```

Loop variable not needed within the loop

Range iterator

- The **range()** function returns an iterator over integers

```
range(from, to, step)
```

- Lower bound is included, upper bound is excluded (as for substring ranges)

```
range(0, 10, 2)
```
→ `[0, 2, 4, 6, 8]`

- If step size is omitted, step is is assumed to be 1

```
range(0, 4)
```
→ `[0, 1, 2, 3]`

- If **range()** is called with one argument, it is interpreted as upper bound

```
range(4)
```
→ `[0, 1, 2, 3]`

- If selected range is empty, iterator does not return any values

```
range(4, 4)
```
→ `[]`

Note: You can use the list constructor to explicitely show the values yielded by an iterator:

```
list(range(4))
```

for loop: break, continue

- The break and continue statements can be also used within a for-loop
  - **break**: Terminates loop execution a continues after loop-block
  - **continue**: Jumps to loop header and iterates over next item

```
for num in range(4, 8):
    if not num % 5:
        break
print("First number divisible by 5:", num)
```

```
Num:  5
```

```
print("All numbers not divisable by 5:")
for num in range(4, 8):
    if not num % 5:
        continue
    print(num)
```

```
4
6
7
```

for loop: else

- The **else** branch of a for-loop is executed, if the loop terminated after having iterated over all elements (and not due to a break statement)

```
for num in range(6, 10):
    if not num % 5:
        break
else:
    print("No multiple of 5")
```

Equivalent code

```
found = False
for num in range(6, 11):
    if not num % 5:
        found = True
        break
if not found:
    print("No multiple of 5")
```

while loop

- Repeats a program block as long a condition is fulfilled

```
while Condition:
    Loop code
```

Use the **while** loop, if the nr. of iterations is **not known** (or is difficult to determine) in advance

- If the condition is not fullfilled (any more), code execution continues after the while-block

```
num = 1
while num <= 20:
    print(num)
    num = num * 2
print("First above 20: ", num)
```

→

```
1
2
4
8
16
First above 20: 32
```

while loop: break, continue

- Execution order in loops can be modified:
  - **break**: terminates loop and continues execution after loop block
  - **continue**: jumps back to loop header and evaluates loop condition again

```
while True:
    answer = input("Do you agree (y/n)? ")
    if answer != "y" and answer != "n":
        print("Invalid answer! Try it again!")
        continue
    if answer == "y":
        print("Good answer, thanks!")
        break
    print("Valid answer, but I don't like it!")
print("Nice that we agree!")
```

Reads console input as string

Endless loop, should be exited via break at some point

while loop: else

- Optional else-branch of a while loop is executed, if the loop execution was aborted due to loop condition becoming False (and not due to a break statement)

```
ii = 0
while ii < 5:
    ii += 1  # ii = ii + 1
    answer = input("Do you agree? (y/n) ")
    if answer == "y" or answer == "n":
        break
else:
    print("Too many invalid answers, I'll assume yes.")
    answer = "y"
print("Your answer was: ", answer)
```