

# Python Programming

## Flow Control

Mihai Lefter



# Introduction

## Outline

Introduction

Sequential Execution

Conditionals

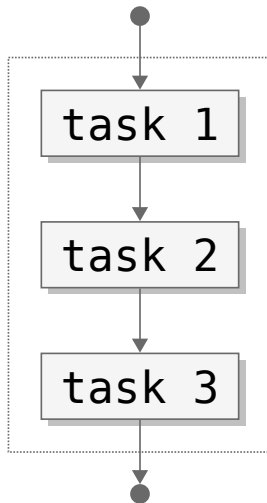
Indentation

Loops

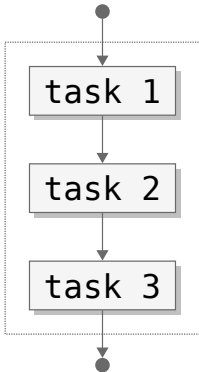
Comprehensions

Hands on!

## Sequential Execution



# Sequential Execution



sum.py

```
1 a = 100
2 b = 200
3 print(a+b)
```

terminal

```
$ python sum.py
300
```

## Sequential Execution

### Intermezzo - User input

Performed with the `input([prompt])` built-in function:

- If the `prompt` argument is present, it is written to the standard output.
- The user input is then read as a string.

# Sequential Execution

## Intermezzo - User input

Performed with the `input([prompt])` built-in function:

- If the `prompt` argument is present, it is written to the standard output.
- The user input is then read as a string.

sum.py

```
1 a = input('a = ')
2 b = input('b = ')
3 print(a+b)
```

# Sequential Execution

## Intermezzo - User input

Performed with the `input([prompt])` built-in function:

- If the `prompt` argument is present, it is written to the standard output.
- The user input is then read as a string.

sum.py

```
1 a = input('a = ')
2 b = input('b = ')
3 print(a+b)
```

terminal

```
$ python sum.py
a = 100
b = 200
100200
```

## Sequential Execution

### Intermezzo - User input

Performed with the `input([prompt])` built-in function:

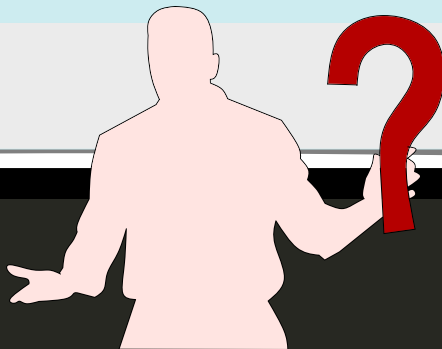
- If the `prompt` argument is present, it is written to the standard output.
- The user input is then read as a string.

sum.py

```
1 a = input('a = ')
2 b = input('b = ')
3 print(a+b)
```

terminal

```
$ python sum.py
a = 100
b = 200
100200
```





# Sequential Execution

## Intermezzo - User input

Performed with the `input([prompt])` built-in function:

- If the `prompt` argument is present, it is written to the standard output.
- The user input is then read as a `string`.

sum.py

```
1 a = int(input('a = '))
2 b = int(input('b = '))
3 print(a+b)
```

terminal

```
$ python sum.py
a = 100
b = 200
300
```

# Sequential Execution

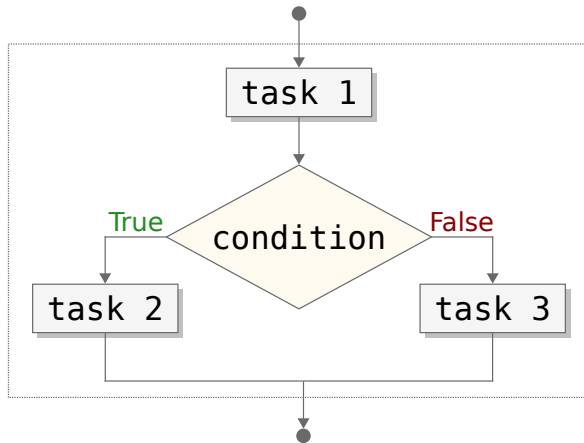
## Intermezzo - Comments

Comments are prepended by `#` and completely ignored.

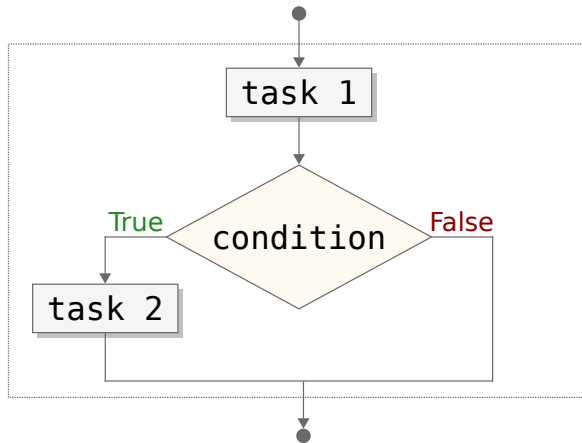
sum.py

```
1  # Retrieve the input
2  a = int(input('a = '))
3  b = int(input('b = '))
4
5  # Compute and display the result
6  print(a+b)
```

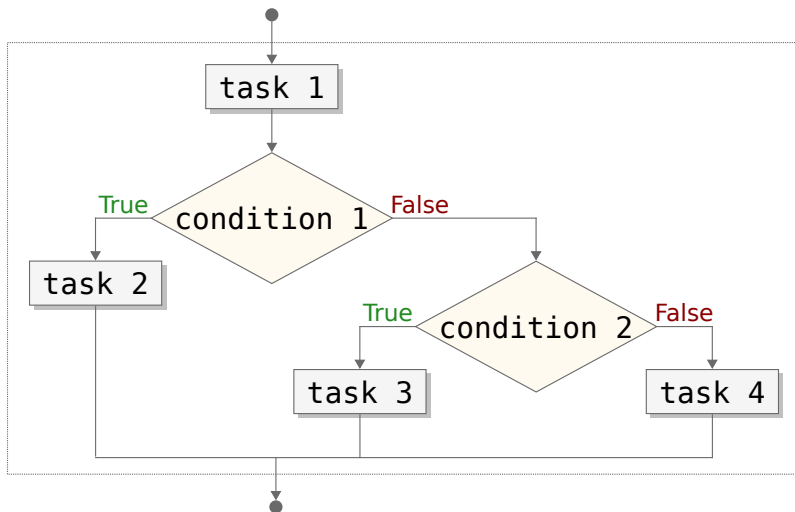
# Conditionals



# Conditionals



# Conditionals



## Truth Value Testing

- Built-in objects considered **false**:
  - constants defined to be **false**: `None` and `False`.

## Truth Value Testing

- Built-in objects considered **false**:
  - constants defined to be **false**: `None` and `False` .
  - zero of any numeric type: `0` , `0.0` , `0j` , `Decimal(0)` , `Fraction(0, 1)` .

## Truth Value Testing

- Built-in objects considered **false**:
  - constants defined to be **false**: `None` and `False` .
  - zero of any numeric type: `0` , `0.0` , `0j` , `Decimal(0)` , `Fraction(0, 1)` .
  - empty sequences and collections: `''` , `()` , `[]` , `{}` , `set()` , `range(0)` .



## Truth Value Testing

- Built-in objects considered **false**:
  - constants defined to be **false**: `None` and `False`.
  - zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`.
  - empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`.
- For the moment, let's assume that any other object is considered **true**.

## Truth Value Testing

- Built-in objects considered **false**:
  - constants defined to be **false**: `None` and `False`.
  - zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`.
  - empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`.
- For the moment, let's assume that any other object is considered **true**.

IPython

```
In [13]: bool(0)
```

```
Out[13]: False
```

```
In [14]: bool(1)
```

```
Out[14]: True
```

```
In [15]: bool([False])
```

```
Out[15]: True
```

## Comparisons

Operation	Meaning	Example
<code>&lt;</code>	strictly less than	<code>x &lt; y</code>
<code>&lt;=</code>	less than or equal	<code>x &lt;= y</code>
<code>&gt;</code>	strictly greater than	<code>x &gt; y</code>
<code>&gt;=</code>	greater than or equal	<code>x &gt;= y</code>
<code>==</code>	equal	<code>x == y</code>
<code>!=</code>	not equal	<code>x != y</code>
<code>is</code>	object identity	<code>x is y</code>
<code>is not</code>	negated object identity	<code>x is not y</code>

# Conditionals

## Comparisons

IPython

```
In [16]: 3 < 4
```

```
Out[16]: True
```

```
In [17]: 3 <= 3.5
```

```
Out[17]: True
```

```
In [18]: 3 == 3.0
```

```
Out[18]: True
```

```
In [19]: 3 is 3.0
```

```
Out[19]: False
```

```
In [20]: 3 is 3
```

```
Out[20]: True
```

### Boolean (Logical) Operations

Operation	Result	Notes
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(3)

1. It evaluates `y` only if `x` is false.
2. It evaluates `y` only if `x` is true.
3. `not x == y` is interpreted as `not (x == y)` and `x == not y` is a syntax error.

## Boolean (Logical) Operations

x	y	x or y	x and y
True	True	True	True
True	False	True	False
False	True	True	False
False	False	False	False

# Conditionals

## Boolean (Logical) Operations

IPython

```
In [21]: 3 < 4 and 5 <= 10
```

```
Out[21]: True
```

```
In [22]: 3 < 4 or 5 <= 10
```

```
Out[22]: True
```

```
In [23]: 3 < 4 and 5 > 10
```

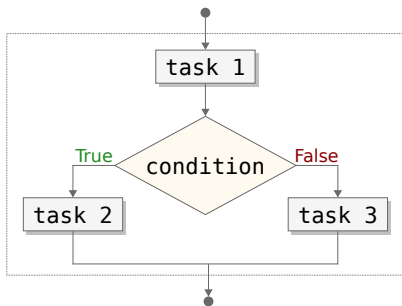
```
Out[23]: False
```

```
In [24]: 3 < 4 and 5 > 10
```

```
Out[24]: True
```

# Conditionals

## `if` statement

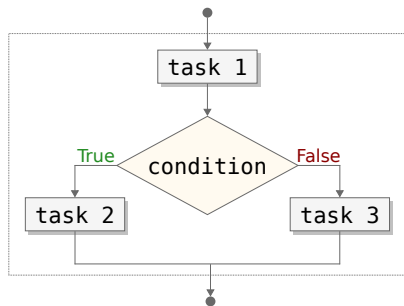


```
if condition :  
    task 2  
else:  
    task 3
```



# Conditionals

## if statement



max.py

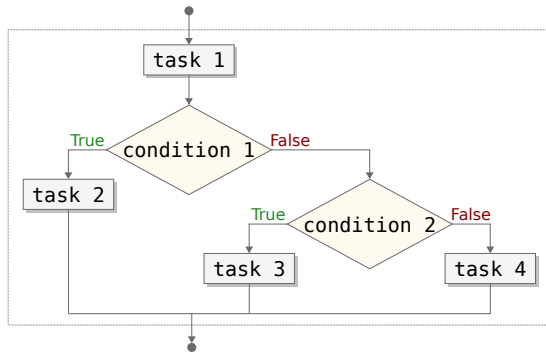
```
1  a = int(input('a = '))
2  b = int(input('b = '))
3
4  if a > b:
5      print(a)
6  else:
7      print(b)
```

terminal

```
$ python max.py
a = 100
b = 200
200
```

# Conditionals

## if statement



compare.py

```
1 a = int(input('a = '))
2 b = int(input('b = '))
3
4 if a > b:
5     print(a)
6 elif a == b:
7     print('equal')
8 else:
9     print(b)
```

terminal

```
$ python compare.py
a = 100
b = 100
equal
```

# Indentation

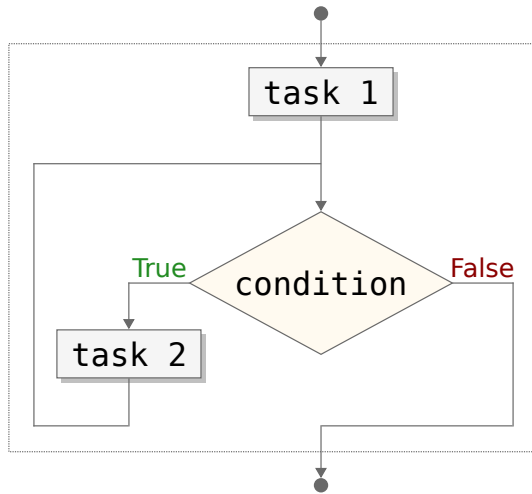
Python uses indentation to delimit blocks

- Instead of `begin ... end` or `{ ... }` in other languages.
- Always increase indentation by 4 spaces, never use tabs.
  - In any case, be consistent.

indentation\_example.py

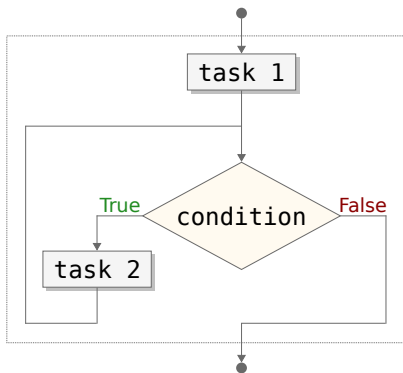
```
1  if False:
2      if False:
3          print('Why am I here?')
4      else:
5          while True:
6              print('When will it stop?')
7  print("And we're back to the first indentation level")
```

# Loops



# Loops

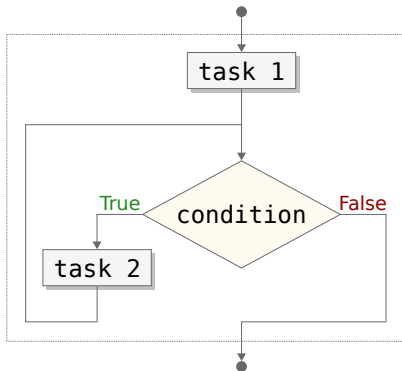
`while` statement



```
while _condition :  
    _task 2
```

# Loops

## while statement



### while\_example.py

```
1 i = 0
2 while i < 5:
3     print(i)
4     i += 1
```

### terminal

```
$ python while_example.py
0
1
2
3
4
```

# Loops

## Infinite loop

infinite\_loop.py

```
1 while True:
2     print('yes')
```

terminal

```
$ python infinite_loop.py
yes
yes
yes
yes
...
```

# Loops

## `for` statement

Used to iterate over a sequence.

for\_example.py

```
1 colors = ['red', 'white', 'blue', 'orange']
2
3 for color in colors:
4     print(color)
```



# Loops

## `for` statement

Used to iterate over a sequence.

for\_example.py

```
1 colors = ['red', 'white', 'blue', 'orange']
2
3 for color in colors:
4     print(color)
```

terminal

```
$ python for_example.py
red
white
blue
orange
```

# Loops

## Python anti-patterns

These are common for programmers coming from other languages.

unpythonic.py

```
1 colors = ['red', 'white', 'blue']
2
3 i = 0
4 while i < len(colors):
5     print(colors[i])
6     i += 1
7
8 for i in range(len(colors)):
9     print(colors[i])
```

We call them unpythonic.

The Pythonic way:

for\_example.py

```
1 colors = ['red', 'white', 'blue']
2
3 for color in colors:
4     print(color)
```

# Loops

`break` and `continue` statements

# Loops

## `break` and `continue` statements

`break` will immediately exit a loop.

break\_example.py

```
1  # Print up to the first negative
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          break
5      print(i)
```

# Loops

## `break` and `continue` statements

`break` will immediately exit a loop.

break\_example.py

```
1  # Print up to the first negative
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          break
5      print(i)
```

terminal

```
$ python break_example.py
6
3
```

# Loops

## `break` and `continue` statements

`break` will immediately exit a loop.

break\_example.py

```
1  # Print up to the first negative
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          break
5      print(i)
```

terminal

```
$ python break_example.py
6
3
```

`continue` will skip the current block.

continue\_example.py

```
1  # Print only positive numbers
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          continue
5      print(i)
```

# Loops

## `break` and `continue` statements

`break` will immediately exit a loop.

break\_example.py

```
1  # Print up to the first negative
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          break
5      print(i)
```

terminal

```
$ python break_example.py
6
3
```

`continue` will skip the current block.

continue\_example.py

```
1  # Print only positive numbers
2  for i in [6, 3, -1, 7, -2, 5]:
3      if i < 0:
4          continue
5      print(i)
```

terminal

```
$ python continue_example.py
6
3
7
5
```

# Loops

## Additional

iteration.py

```
1  # Iteration with values and indices:
2  for i, color in enumerate(['red', 'yellow', 'blue']):
3      print(i, '->', color)
4
5  # Taking two sequences together:
6  for city, population in zip(['Delft', 'Leiden'], [101030, 121562]):
7      print(city, '->', population)
8
9  # Iterating over a dictionary yields keys:
10 for key in {'a': 33, 'b': 17, 'c': 18}:
11     print(key)
12
13 # Iterating over a file yields lines:
14 for line in open('data/short_file.txt'):
15     print(line)
```



# Loops

## The `pass` statement

If you need a statement syntactically, but don't want to do anything yet, use `pass`:

pass\_statement.py

```
1 age = int(input('Please enter your age: '))
2
3 if age < 18:
4     # This is to be decided.
5     pass
6 else:
7     print('You can apply for a driver\'s permit in most of the countries.')
```

# Comprehensions

## Lists

Similar to mathematical set notation (e.g.,  $x|x \in R \wedge x > 0$ ), we can create lists.

IPython

```
In [25]: [(x, x * x) for x in range(10) if x % 2]
```

```
Out[25]: [(1, 1), (3, 9), (5, 25), (7, 49), (9, 81)]
```

# Comprehensions

## Sets and dictionaries

IPython

```
In [26]: {c for c in 'LUMC-standard' if 'a' <= c <= 'z'}
```

```
Out[26]: 'a', 'd', 'n', 'r', 's', 't'
```

# Comprehensions

## Dictionaries

IPython

```
In [27]: colors = ['red', 'white', 'blue', 'orange']
```

```
In [28]: {c: len(c) for c in colors}
```

```
Out[28]: {'blue': 4, 'orange': 6, 'red': 3, 'white': 5}
```

## Hands on!

Write a python program for each of the following exercises:

### 1. **Special numbers**

Print only those numbers which are divisible by `13` and multiple of `5`, between `10` and `1313` (both included).

### 2. **Bank account simulator**

Take as input an initial bank account balance (e.g., `1000`). Next, accept inputs consisting of either expenses (e.g., `-13.99`) or revenues (e.g., `20`) until the user introduces `exit`. After each input, in case the balance is about to go negative, print an error message (e.g., `Operation not permitted: insufficient funds.`). Otherwise, print the new balance value.