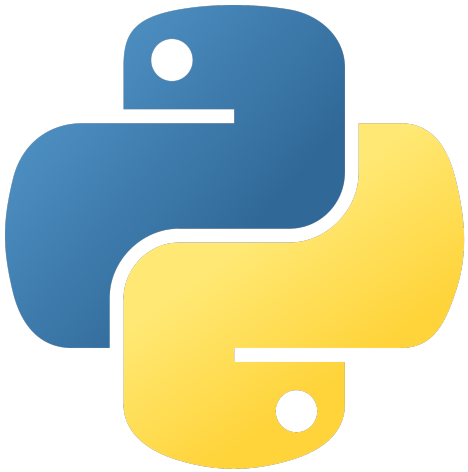
Introduction to Python3

An introduction to Python part 1 – Beginner



Christoph Rosemann

DESY FS-SC

November 27, 2025

Front matter

What is this about?

- central idea: gentle hands-on introduction
- by choice very reduced amount of topics
- education on your terms, please interact/ask
- as much (inter-) activity as possible
- structured into blocks
- target for today: being able to read and write simple programs
- target for tomorrow: have a good basic idea of Python
- this is a work in progress, first iteration!
- several continuations possible, evaluation of the approach

Outline for today

Topics

- introduction
- data types and data structures
- statements and expressions
- control flow
- input/output

Style

please engage!

Next

Block::Introduction

too much talking, but necessary

use of colours:

blue: general text

green: things to type, to test or to do

red: important things to look out for

keywords (reserved words) are in different font

Python characteristics

Python is

- a strongly typed language
- a dynamically typed language
- multi paradigm language
- set in blocks that are identified by indentation
- interpreted language (relying on byte code)

Lots of humour and quotes surrounding it

- "Coding like Guido indented it"
- "Walks like a duck, quacks like a duck – is a duck"
- "Batteries included"

Python?

it's named after Monty Python

Python Interactive

scripts? program? code!

There are several ways to write and execute Python code or programs.

- interactive prompt
- editor and command line
- integrated development environment IDE

Most simple view: a Python program is just a file containing Python statements.

Python Interactive

for now: interactive prompt
indicated by >>>
note: it prints the results of every expression!

Tradition first: the first program

Python Interactive

```
>>> print("Hello world!")
```

Write program file

- start editor
- enter
  ```
  print("Hello world!")
  ```
- save file as 'hello.py'
- execute as `python3 hello.py`

Built in command print()

takes an argument (since it is actually a function, see later)

Help and references

The best source:

- `https://docs.python.org/`

Books:

- Mark Lutz *Learning Python*, ed. 6 recently published
- Mark Lutz *Programming Python*, ed. 4

Python Interactive:

- `>>> help(`*object*`)`
- `>>> dir(`*object*`)`

Python is interactive!

Central point about programming

Programs are meant to be read by humans and only incidentally for computers to execute.

(Abelson & Sussman, "Structure and Interpretation of Computer Programs")

Python Interactive: Folklore

```
>>> import this
```

A preview of terms

We will return later to these

A hierarchy of names

1. programs can be/are composed of modules
2. modules contain statements
3. statements contain expressions
4. expressions create and process "objects"

Things for the back of your mind

- in Python, everything is an object
- programs and modules are just text files
- in Python, (almost) everything is a reference

Next

Block::data types and data structures

What are the "things" we are doing "stuff" with?

Topics

- data types in Python
- basic data types: numbers, strings, and booleans
- built-in data structures: lists, dictionaries, tuples (and more)
- from simple data types to categories (ie. sequences and mappings)

What is a data type?

Definition

A data type defines what kind of value a variable can hold and what operations can be performed on it.

Examples using the assignment operator =

```
>>> x = 42                          (integer)
>>> y = 3.14                        (floating-point)
>>> name = "John Cleese"            (string)
```

From now on: drop the >>> in front of the line

Built-in objects

A non exhaustive overview

Numbers	1234 , 3.1415 , 3+4j , Decimal , Fraction
Strings	'spam' , "eggs"
Lists	[1, [2, 'three'], 4]
Dictionaries	'food': 'spam', 'taste': 'yum'
Tuples	(1, 'spam', 4, 'U')
Files	myfile = open('eggs', 'r')
Sets	set('abc'), 'a', 'b', 'c'
Other	Booleans, types, None
Program unit types	Functions, modules, classes

Incomplete view in more than one way

Central concepts will become apparent later:
Categories, Immutability, Mutability, ... go step by step

Numbers

The most simple/straightforward type

- `int` — whole numbers
- `float` — decimal numbers
- `complex` — numbers with real and imaginary parts

Another function: type()

- built-in function ($\rightarrow$ later)
- typically not used in Python code
- here used for illustration
- can be helpful in figuring out unexpected behaviour

Numbers interactively

Try in Interpreter

```
a = 7
b = 2.5
print(a + b)
print(type(a), type(b))
print(type(a+b))
k = 2+4j
print(type(k))
pow(a, 2)
a ** 2
abs(k)
```

comment: more functions, for now they

- are objects with a reserved name
- take one or multiple arguments given in brackets

Numbers

Still a non exhaustive overview

1234, -24, 0, 9999999999	Integers (unlimited size)
1.23, 3.14e-10 , 4E210 , 4.0e+210	Floating-point numbers
0o177 , 0x9ff , b'101010'	Octal, hex, and binary literals
3+4j , 3.0+4.0j , 3J	Complex number literals
True, False	Booleans

Each type has a clear identifier

no dot, dot, leading prefix, special character

There are extensions to this

example: fractions (rational numbers), vectors, matrices, ...

Strings

Strings in a nutshell

- single or double quotes enclosure
- multi-line with triple quotes enclosure
- ordered sequence of characters
- immutable (cannot be changed in place)

Strings are a massive topic on their own

- regular expressions
- handling input
- all kinds of supports
- special importance, since users like formatted text
- we skip most of that for now!

Strings interactively

Examples

```
s = "abcdefghijkl"
t = 'strings are a big topic'
u = '''one could possibly say that
       none of the many many
       possibilities are
       boring?'''
print(s[0], s[-1])
print(u)
print(len(s))
print(s.upper())
```

Why are there different ways to construct?

Booleans and None

more special basic types

- `bool` represents truth values: `True`, `False`.
- Often the result of comparisons.
- `None` is a special type representing “no value”.

Example

```
temperature = 25
print(temperature > 20)
result = None
print(type(result))
```

Conclusion

These are the basic types: int, float, str, bool, None

Why Data Structures?

The power of structure

- real data is often a collections of values
- data structures group related data and let us process it efficiently
- Python provides several built-in structures
- these are at the core of what makes Python both user friendly and extremely powerful

Do I need to care about types? Sometimes YES!

- most of the time you'll be using variables without every thinking about their type(s)
- even more times you'll be using data structures to solve specific tasks

Let's have a closer look

Lists: Ordered and Mutable

Lists – the ultimate super power of Python

- ordered sequence of values
- constructed with square brackets, elements are comma separated
- empty list constructed by empty square brackets
- can hold any (!) data type and structure
- can be accessed by position or by an iterator
- mutable: can be changed in place
- central idea: lists are sequences that can grow or shrink
- `list` is a reserved word

There is a twist to lists!

Less dramatic: one has to be aware of a property in using them

Lists interactively

Examples

```
data = [2, 4, 6, 8]
print(len(data))
data.append(10)
print(data)
data[3] = 10
print(data)
beta = data
delta = data.copy()
print(beta)
beta[0] = "hello"
print(delta)
print(data)
```

Lists caveat

create duplicates of lists by using the `copy()` function

Be aware when "copying" lists

- possibly a longer excursion, simplified take away:
  new assignment of a list to another variable is the identical list!
- unfortunately this behaviour seems different in other cases

Tuples: Ordered and Immutable

Tuples

- constructed with round brackets, elements are comma separated
- empty tuple constructed by empty round brackets
- can hold any (!) data type and structure
- can be accessed by position or by an iterator
- useful for fixed collections of data (e.g., coordinates, parameters)

Example

```
point = (3, 4)
print(point[0])
# point[0] = 5  # This would raise an error
```

Dictionaries: key–value pairs

Dictionaries

- constructed with curly brackets, each entry being a pair of key and value separated by a double colon, multiple elements are comma separated
- empty dictionary constructed by empty curly brackets
- always one key that is associated with a value
- is an associative container: get values by key, not (!) position
- keys must be immutable, other than that any type or structure is allowed
- can hold almost any (!) data type and structure as value
- different iterators or sub collections exist: `items()`, `keys()`, `values()`

Dicts share the list property

... and the solution

Dictionaries interactively

Examples

```
d = {}
type(d)
measurements = {"day1": 23.1, "day2": 22.8}
print(measurements["day1"])
measurements["day3"] = 24.0
```

Sets: Unique Unordered Elements

Sets

- constructed with curly brackets, elements are comma separated
- each element is unique (!), duplicates get eliminated
- cannot construct an empty set
- typical use: membership tests and eliminating duplicates.

Example

```
samples = {"A", "B", "C", "A"}
print(samples)
print("A" in samples)
```

Type categories – and the Pythonic perspective

Pythonic perspective

- Python uses types, but the experience is different
- types and structures are defined by the operations they support
- some operation might seem "intuitive", ie. multiplication or summation

Categories

- numbers: int, float, fraction, ...
- sequences: strings, lists, tuples
  - ordered (!) collections of variables
  - can be accessed by position
- mappings: dictionaries – key-value pairs

Changeability

To change or not to change?

Orthogonal property to which category a type belongs to

- deliberate design choice
- possible variation within the same category (eg. `list`/`tuple`)

Immutables

numbers, strings, tuples, frozen sets

- no change possible
- must be set at creation time

Mutables

lists, dictionaries, sets, byte array

- very powerful feature: change in place
- potentially very dangerous feature!

Block summary

types, structures and categories

- Python has simple data types: numbers, strings, booleans
- built-in data structures: lists, tuples, dictionaries, sets
- data types form categories: sequences, mappings, sets

Next

Block::building blocks – statements and expressions

Repeated from before:

- programs are composed of modules
- modules contain statements
- statements contain expressions
- expressions create and process objects

Topics

- introduce the building blocks of a Python program
- distinguish between expressions and statements
- start to combine data and operations into meaningful programs

What is a Program?

Definition

A program is a sequence of instructions that the computer executes to perform a task.

In Python

- each instruction is written as a statement
- statements often contain one or more expressions

Example

```
area = width * height
```

- statement: assignment to `area`.
- expression: `width * height`.

Expressions

What is an expression in Python?

- an expression is something that produces a value
- the interpreter can evaluate it

try in interpreter

```
2 + 3
"Hello" + " " + "World"
len([1, 2, 3])
3.14 * (2 ** 2)
```

update on the previous comment:
expressions display their result in an interactive session

Statements

In Python

- a statement performs an action
- examples:
  - assign a value to a variable
  - print output
  - import a module (see later)
  - create a loop or conditional (see next block)

Examples

```
x = 5           # assignment statement
print(x)        # function call statement
import math     # import statement
```

Expressions vs. Statements

Python summary

	Expression	Statement
Produces a value?	Yes	Not necessarily
Can stand alone?	Sometimes	Always
Examples	`2 + 3`, `"A" * 3`	`x = 5`, `print(x)`

Takeaway

- be aware what a code snippet does, ie. return something
- how that is called is less important

interpreter behavior

once again, in an interactive session:

- expressions return a value
- statements do something but don't return a value

Combining Statements and Expressions

Example Program

```
radius = 3.0
area = 3.1416 * (radius ** 2)
print("Area of circle:", area)
```

in this example:

- each line is a statement
- `3.1416 * (radius ** 2)` is an expression
- together, they form a small but complete program

Interactive Practice

try

in the Python interpreter:

1. compute the kinetic energy of an object:
   `E = 0.5 * m * v ** 2`
2. assign variables `m` and `v` and print the result
3. add a new statement that doubles the velocity and recomputes

discussion:

What changes if we enter only the expression `0.5 * m * v ** 2` versus the full assignment?

Next

Block::control flow

What makes a program a program?

topics

- understand how Python decides what to execute next
- learn to use `if`, `elif`, and `else`
- use loops (`for`, `while`) to repeat actions
- combine control flow with expressions and statements to build small programs

Programs Need Choices and Repetition

How to write a program

- until now, statements executed only sequentially
- real programs:
  - make decisions (e.g., is a measurement valid?)
  - perform actions repeatedly (e.g. analyze all samples, values, events)
- Control Flow means deciding and repeating

if Statements

General Form

```
if condition:
    # block of code
elif another_condition:
    # optional block
else:
    # optional fallback
```

Indentation

- first case we see that: defines the block structure in Python
- visual clue in code what parts belong together
- in interpreter: done automatically
- in a file: customary to use 4 spaces
- attention: don't mix tabs and spaces, always use the same indentation!

Example: Temperature Check

Try in Interpreter

```
temperature = 37.5

if temperature > 37:
    print("High temperature")
elif temperature < 35:
    print("Low temperature")
else:
    print("Normal range")
```

Discussion:

what happens if `temperature = 35.0`?

Comparison and Logical Operators

Operators: character combinations with special meaning

- Comparison: ==, !=, <, >, <=, >=
- Logical: `and`, `or`, `not`

Example

```
if (temp > 20) and (humidity < 80):
    print("Comfortable conditions")
```

very common mistake

don't confuse assignment = with comparison ==

```
  if temp = 20:
```

versus

```
  if temp == 20:
```

`for` Loops: Iterate Over a Sequence

Example

```
samples = ["A", "B", "C"]

for s in samples:
    print("Analyzing sample", s)
```

Idea: The loop variable `s` takes each value in the sequence.

Extra knowledge: iterator

This is another high level concept at work in Python: the iterator. It's an object, which visits every entry of a data structure when used in loop.

for loops with iterators

Examples

```
data = [2, 4, 6, 8]
for value in data:
    print(value ** 2)
for k in measurements.keys():
  print(k)
for k, v in measurements.items():
  print(" the key is:", k, "and the value is", v)
```

`range()` for Numerical Loops

Example

```
for i in range(5):
    print("Measurement", i)
```

range(): a special kind of function

- `range(n)` generates numbers from 0 to n-1
- useful for repeating a fixed number of times
- try: what is the type of `range(n)`?
- makes sense only in a loop context

`while` Loops: Repeat Until Condition Fails

Example

```
energy = 10
while energy > 0:
    print("Energy:", energy)
    energy -= 3
```

eternal loops

Be careful! Loops must eventually make the condition false, or they run forever.
This might be wanted, but how does one get out?

Loop Control: `break` and `continue`

Two more keywords to control loop behaviour

- `break` — exit the loop early.
- `continue` — skip to next iteration.

Example

```
for value in [3, -1, 5, 0, 2]:
    if value < 0:
        continue
    if value == 0:
        break
    print("Positive:", value)
```

Interactive Practice

Try in the interpreter:

1. Write an `if` statement that classifies a number as positive, negative, or zero.
2. Create a `for` loop that prints the square of numbers 1–5.
3. Modify it to skip even numbers using `continue`.

Question: When would you prefer a `while` loop over a `for` loop?

Block Summary

control flow

- `if/elif/else` choose what code to run
- `for` and `while` loops repeat code
- control flow connects statements into logical programs