# Python

## Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

## Three Families of Higher Programming Languages

We have have seen that can be classified into three groups: imperative, functional, and declarative.

Python is an imperative language.

It has dynamic typing. (Each value has a type at runtime, but variables themselves do not have a type.)

Python uses an interpreter.

Python uses reference semantics. Its interpreter uses garbage collection.

Python supports Object-Oriented Programming.

Python does not guarantee backward compatibility.

Python recently added optional static type checking (3.09)

## History

Python was developed by Guido van Rossum at Centrum voor Wiskunde en Informatica in Amsterdam in 1989.

Before that, he had worked on another language called ABC, which seems to have failed.

He designed Python, based on the lessons learnt during ABC.

He carried the title 'Benevolent Dictator for Life (BDFL)' from 1995 until July 12th 2018.

## Resources

I mostly use online resources:

`www.geeksforgeeks.org/python-programming-language/`

`https://www.w3schools.com/python/`

`https://docs.python.org`

## Interpreted Language

In compiled languages like $C$ or $C^{++}$, the program is translated into machine instructions before it can be executed.

This is efficient at run-time, but you have to wait for the compiler, and it is hard to see internal state during testing.

The specification of some parts of the language (e.g. size of `double` or `int`) are left to the machine or the compiler.

Python uses an interpreter. The interpreter reads the program, translates it into an internal machine indepenent representation (a stack machine) and runs this stack machine.

The internal representation preserves more information about the source, type checking can be postponed, but execution is slower.

Java does something inbetween.

## Reference vs. Value Semantics

$C^{++}$ is based on value semantics. Assignment (+ initialization + parameter passing) by default means : copying the complete value. Memory lay out is simple: Fields and array elements are next to each other in memory. CPUs like predictable memory use.

Python and Java use reference semantics. Assignment (+ initialization + parameter passing) means: Copying the pointer (reference) to the data. As a result, data is shared, consecutive fields and array elements are not close in memory. Ownership is unclear, there are side effects, and one needs a garbage collector.

At runtime, many cache misses and compiler cannot optimize.

In memory and CPU registers are only bits and bytes which don't have a type. Machine instructions assume a type (addint, addfloat, addunsigned).

One has to make sure that the instructions fit to the intended type. For example storeint R0, 1000 ; / loadfloat R1, 1000 is meaningless.

static type checking: Decide the types at compile time, create the correct machine instructions, and forget about types at run time.

dynamic type checking: Attach type information to the data, and use these data to select the proper instruction at run time.

Dynamic type checking is sometimes called duck typing: "If it walks like a duck and it quacks like a duck, then it must be a duck"

(In its most extreme form, you forget about types at all, and only remember which operations are allowed.)

In order to start the interpreter, type `python3`.

It asks for input, and responds to the input. Commands are executed immediately when you type them:

```
1 + 1  # prints 2


a = 1
b = a + a + a
b      # prints 3.
```

## Interpreted

You can type

```
a = 4
b = 8
a + b     # Prints 12.

while a > 0 :
   print (a)
   a = a - 1

   # Prints 4 3 2 1
```

As you see, you can type programs, which are executed at once.

If you want to define a function, type:

```
def print4321( ) :
    a = 4
    while a > 0 :
        print(a)
        a = a - 1
```

Remember that the Python interpreter is interactive. Function definition is a kind of assignment, where the function body is assigned to variable `print4321`.

In reality, typing long programs at once is unpractical.

It is better to prepare the program in a text file, and load the file. In file **hello.py**, write:

```
def sayhello( ) :
    print( "hello" )
```

On the command line, type:

```
from hello import *
sayhello( )
```

or

```
import hello
hello. sayhello( )
```

If you import in the first way, the function will be called just `sayhello( )`.

If you import in the second way, the function is called `hello. sayhello( )`.

## Importing

The same applies when you load a library, for example `scipy`

```
from scipy import *
   # Everything can be used without prefix:
m1 = array( [ 1, 2 ] )

import scipy
   # Everything in scipy has to be prefixed by
   # 'scipy':
m1 = scipy. array( [ 1, 2 ] )
```

You can also load a module giving it another name, if you want to use another name:

```
import scipy as sp
m1 = sp. array( [ 1, 2 ] )
```

If inside some file you import another file, the prefixes add up. This makes it sometimes hard to find a function. If you want to know whether a function exists, you can just try to print it.

```
somefunction
# Prints an address if the function exists,
# otherwise an error
```

## Advantages of Interpreted Language

You can type commands at the command prompt, and get immediate answer:

```
a = 3
b = 4
a*a + b*b
a = 5
b = 12
a*a + b*b
```

You don't have to wait for the compiler, and you don't have to start an editor.

## Interpreted Language (2)

It is easy to test your program, because you can interactively set variables, call functions, and check the values of variables.

In other languages, you have to change `main`, put print statements, and recompile.

Disadvantages:

It is slow in production code.

Your code cannot run without the interpreter.

Your client sees your source code.

With bigger programs, you need a separate editor anyway. Loading becomes slower.

The interpreter stops on the first error. A compiler usually reports all of them at once.

## Advantages of Dynamic Typing

It is flexible. You get polymorphism (functions working on different types) for free. For example,

```
def square(x) :
   return x * x



   square(4)   -->   16
   square(4.0)  -->  16.0
   square( rational.Rational(1,2) ) -->  1/4
```

## Disadvantages of Dynamic Typing

Every time when function `square` is called, the interpreter must decide if `x` has a `*`-operator, and select the right one. Finding a suitable `*`-operator takes more time than the multiplication itself.

It is possible that the interpreter does not find a `*`-operator, which means that there can be errors at run time.

For example, define

```
def verygoodmorning( ) :
    return square( "good morning" )
```

This program can be imported without problems, and gives an error only when you call it.

In a statically typed language, the compiler selects the proper * at compile time, and produces an error if none is found.

During execution, only calculations are done, no decisions and checks are made.

It is guaranteed that the program will not crash.

In $C^{++}$, if a polymorphic function (template) is used with three types, the compiler compiles three versions.

## Problems with Integer Division

Python has some difficulties with integer division. Before version 3.0, the `/`-operator worked as in *C* :

On integers, `/` uses integer division, on `double`, `/` uses double division. This results in:

```
10 / 3   -->   3
10.0 / 3 -->   3.3333333
```

This works strange with dynamic typing, since type of `x` in `x/3` can vary.

Therefore, in version 3.0, the meaning of `/` was changed to always create a `double`.

If you want integer division use `//`.

In this course, you must use at least Python 3.12.

## Backward Compatibility

Other languages, (most notably $C^{++}$ and Java) guarantee backward compatibility.

A program that works at some point in time, will work with later versions.

Python does not support backward compatibility. The switch from Python 2 to Python 3 is legendary.

## Backward Compatibility (2)

Advantages: You don't have to worry about compiler versions. If your program works on some compiler, it will work with later versions. This is very important in big projects that last over many years. You can upgrade your compiler without fear.

Disadvantages: The language becomes conservative. Because it is impossible to remove something, language designers become afraid to add things to the standard.

Programmers keep on using constructions that they should not use anymore. (naked `new` and `delete`, `malloc` and `free` in $C^{++}$). Small errors accumulate.

The risk is that programmers switch to a completely different language e.g. $C^{++} \Rightarrow$ Rust.

## Indentation

Python enforces indentation. I repeat:

# Python enforces indentation!

On the positive side, there are no `{` and `}` in Python, because scope is controlled by indentation.

Note that you cannot mix TABS and spaces in indentation.

## Complex Numbers

In addition to `double`, Python supports complex numbers.

```
1j * 1j      # prints -1 + 0j

2 / ( 1 + 1j )  # Prints ( 1 - 1j )
                # 2 is not prime in Gaussian integers.

import cmath
cmath.exp( cmath.pi * 1j )
   # Prints a number close to -1.

   # Note that floating point calculations have
   # rounding errors.
```

We will calculate $\sin(z), \cos(z), e^z$ using the Taylor series:

$$e^z = \sum_{i=0}^{\infty} \frac{z^i}{i!}.$$

$$\sin(z) = \sum_{i=0}^{\infty} (-1)^i \frac{z^{2i+1}}{(2i+1)!}$$

$$\cos(z) = \sum_{i=0}^{\infty} (-1)^i \frac{z^{2i}}{(2i)!}.$$

The sequences converge everywhere.

## Reference Semantics

You all remember that $C^{++}$ has value semantics. This means that assigning is copying, and distinct variables are usually unrelated.

Python has reference semantics:

```
a = [ 1, 2, 3, 4 ]
b = a
a[1] = 100;
b     # is also changed
```

## Use of `is`

If you want to know if two variables are the same object (have the same representation), use `is`.

```
A = [1,2,3,4]
B = A
A is B   # True

B = [1,2,3,4]
A is B   # False
```

## Use of `id`

You can also use `id( )`, which prints the address:

```
A = [1,2,3,4]
B = A
id(A)    # some number
id(B)    # same number

B = [1,2,3,4]
id(B)    # another number
```

## Immutability

Since reference semantics can be fairly dangerous due to side effects, Python also has *immutable objects*.

It is a bit similar to **const** in $C^{++}$ but not the same. An object is immutable if there exists no operation that change any of its components in place. This is a property of the object itself.

At the same time constness is a property of the variable referring to the the object. A const variable forbids access to methods that change the object in place, and it forbids replacing the object as a whole.

We have seen a few slides back that **list** in Python is mutable. **scipy.array** is also mutable.

**string**s and **tuple**s are immutable.

```
S = "good morning"
S[0] = 'G'  # Error
```

```
S = (1,2,3)
S[1] = -2   # Error.
```

## Immutability versus Constness

constness is a property of the variable referring to the object. In case different variables refer to the same object, one may be const, while another one is not const. One cannot change an object through a const variable, and one also cannot replace it.

immutability is a property of the object. It means that there exists no operation that modifies part of the object in place. It is possible to assign to a variable that contains an immutable object. In that case, the complete object is replaced.

Java uses final to prevent replacing the object as a whole. This is again a property of the variable.

In Python, variables have no properties. Only data has.

## Tuples

Tuples are like lists, but immutable. They can be written with round parentheses `( )` or with commas only:

```
A = 1,2,3
A
A = 1,  # Tuple with one element
A
```

Tuples can be used if one wants to return many values from one function, and they can be used for assigning multiple variables at once.

Exchanging two variables:

```
a,b = 1,2
a,b = b,a
```

## GCD-algorithm:

```
def gcd( n1, n2 ) :
    if n1 == 0 and n2 == 0 :
        raise ArithmeticError( "gcd(0,0) does not exist" )
    while n2 :
        n1, n2 = n2, n1 % n2
    return n1
```

## Fibonacci numbers:

```
def fib( n ) :
   if n == 0 :
      return 0

   f1,f2 = 0,1
   for i in range(1,n) :
      f1, f2 = f2, f1 + f2
   return f2
```