# Intelligent Control
# Lecture 2 – Data types, Variables and Sequences

UNIVERSITY OF APPLIED SCIENCES

Hanshu Yu

30th Nov. 2022

De persoonlijke hogeschool

## Contents

UNIVERSITY OF APPLIED SCIENCES

## Python Data Types

You do not need to declare variables in Python.

**Dynamically** typed

int, float, str, bytes, list, tuples, dictionary, set, Boolean, file, NoneType

## Dynamically typed vs Statically Typed

Do type checking at runtime. (types are not associated with variables)

Do type checking before runtime. Typically at compile time. (compile C file: TypeERROR)

## Python Numbers

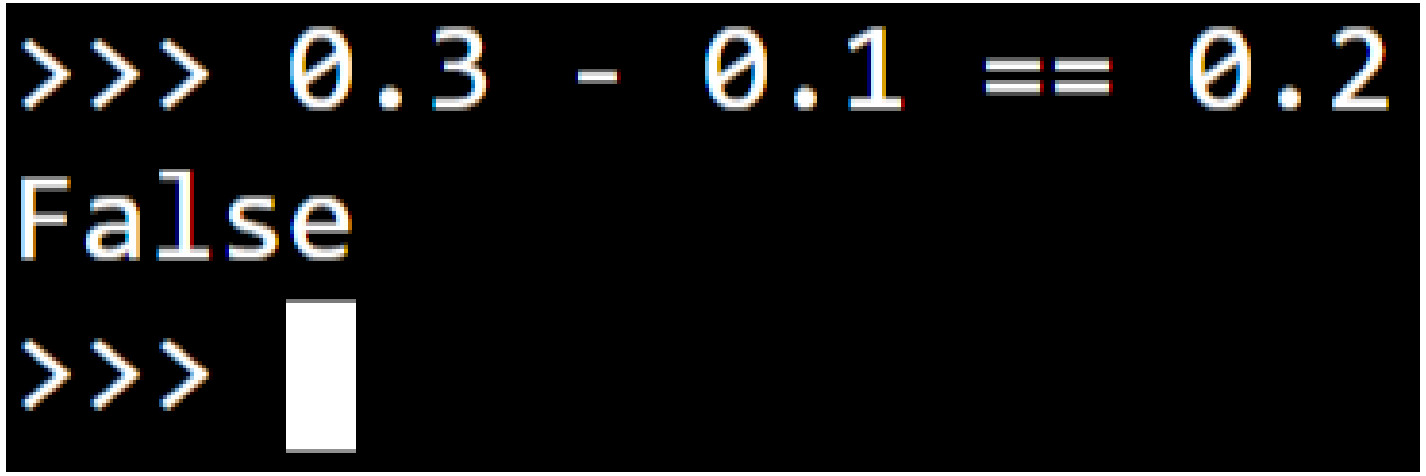
Python store numbers in binary, with a variable number of bits. (depends on the memory available)

| | |
|---|---|
| Integers | 1, -1, 0b10011010(binary), 001247(octal), 0x3ad7 (hex) |
| Floating point | 1.0, 1.2345, 2.71828e-25, 8e12 |
| Complex numbers | 3+4j, 3j, 3.14–2.71j |

Dynamically ‘stretched’ to the upper most compatible type

## Something interesting

## Something interesting



```
>>> 0.3 - 0.1 == 0.2
False
>>> 
```

## Something interesting

$$0.1 = \frac{1}{10}$$

In binary

$$\frac{1}{1010}$$

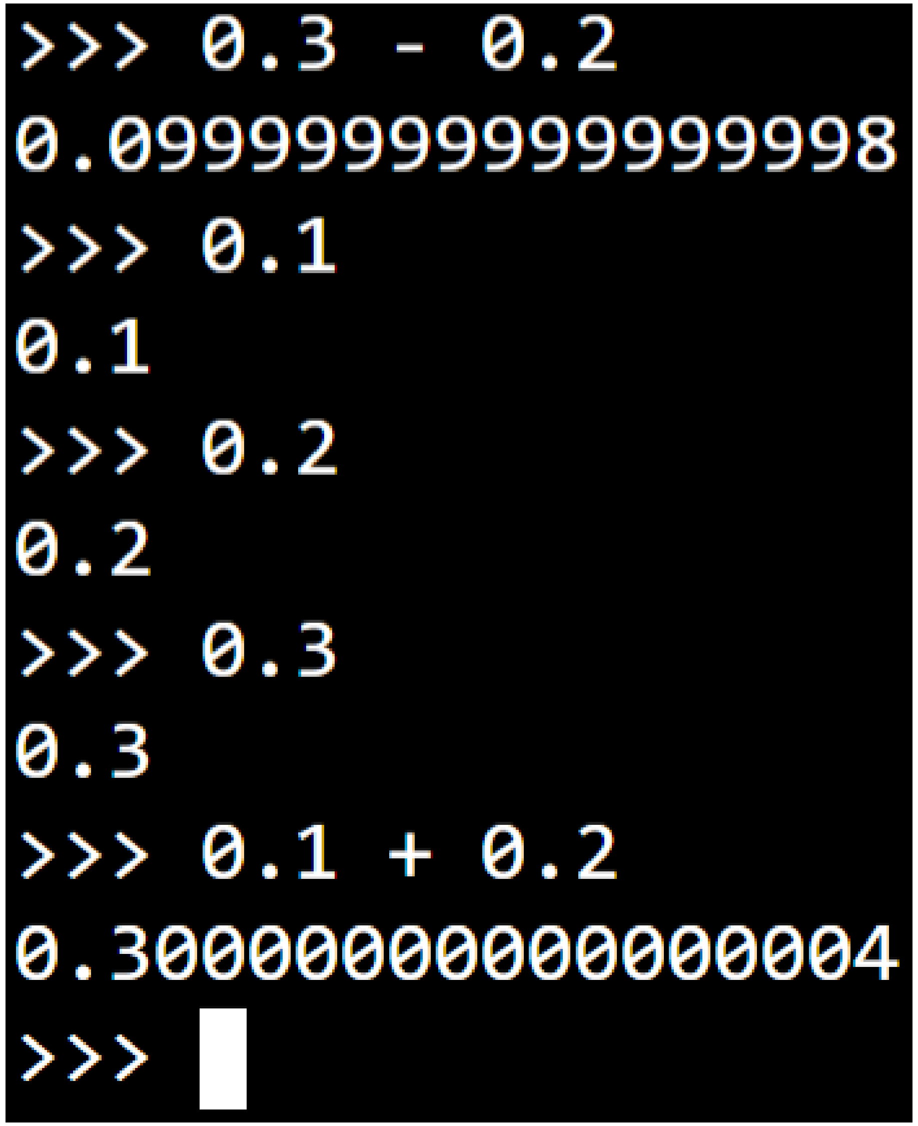
(Binary)Division:

1/1010 = 0.0001001001001001001……

= $0.0\underline{001}$

## But, hey! Wait a minute…

## Something interesting



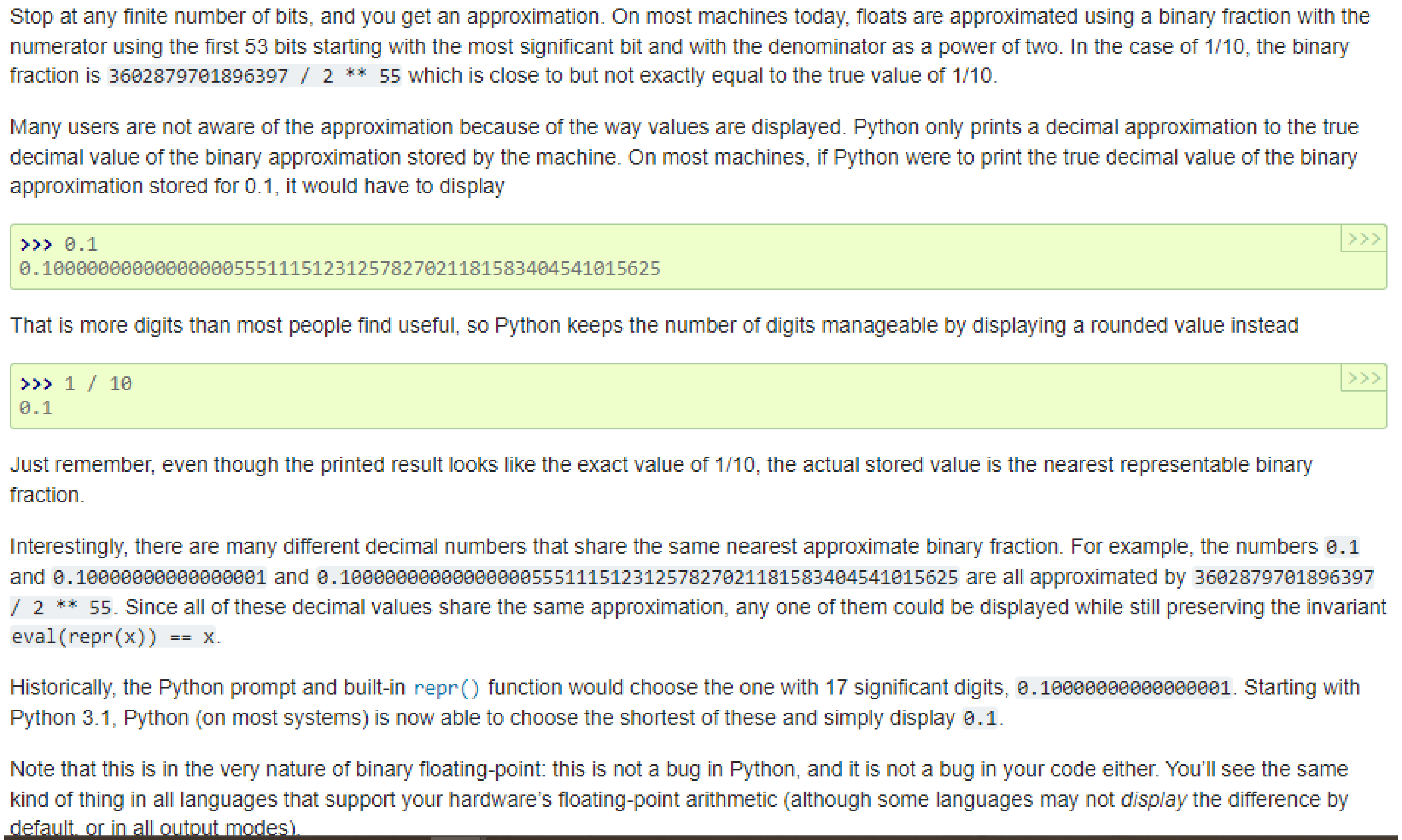
```
>>> 0.3 - 0.2
0.09999999999999998
>>> 0.1
0.1
>>> 0.2
0.2
>>> 0.3
0.3
>>> 0.1 + 0.2
0.30000000000000004
>>> 
```

How come the Python interpreter know 0.1 exactly when I call it?? Isn't that 0.1 exact?

## Why is that?????

https://docs.python.org/3/tutorial/floatingpoint.html

## Something interesting – The Truth



Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of 1/10, the binary fraction is `3602879701896397 / 2 ** 55` which is close to but not exactly equal to the true value of 1/10.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead

```
>>> 1 / 10
0.1
```

Just remember, even though the printed result looks like the exact value of 1/10, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers `0.1` and `0.10000000000000001` and `0.1000000000000000055511151231257827021181583404541015625` are all approximated by `3602879701896397 / 2 ** 55`. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, `0.10000000000000001`. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display `0.1`.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

## Python Strings

‘This is a piece of python string’

“This is a piece of python string”

“‘ You can use “ in ‘ and ’ ”’

‘Or use black slash like \’ ’

‘Double quotes “” can be printed inside single quote’

“Single quotes ‘’ can be printed inside double quote”

“ “ “String with
multiple lines
” ” ”

UNIVERSITY OF APPLIED SCIENCES

## Python Identifiers

What is an identifier in python?

simply: names of variables, functions, classes, class instances, etc.

- Be aware of the reserved keywords of Python. (You can find the complete list easily in our textbooks.)
- Case sensitive
- Should not start with numbers
- No special symbols allowed, including blank spaces
- No length limitation

UNIVERSITY OF APPLIED SCIENCES

## Python Strings

Python str use Unicode characters by default

Default output encoding: UTF-8

(but sometimes depends on your exact machine & OS settings)

Use backslash \ as escape characters

Or use the r-prefix in front of the string text

Other prefix: u, r, b, f (starting from Python 3.6)

Use encode(), decode() methods to alter the encodings

https://docs.python.org/3.11/library/codecs.html#standard-encodings

## Python Operators

**Arithmetic operators:**

+ add,
- subtract,
* multiply,
** power,
/ divide,
// truncate,
% modulo,
@ matrix multiplication,

**Bitwise binary operators:**

^ XOR bitwise addition,
| OR 0 if both 0 otherwise 1,
& AND – bitwise multiplication,
~ complement,
>> shift left bit,
<< shift right bit,

**Comparison operators:**

>, <, =, ==, !=, >=, <=

## Python Operators

Logical operators: and, or, not

Identity operator: is, is not

Membership Operators: in, not in

## Demo

Time to stretch!

Leave your desk!

## Python Sequences and Indexing

Built-in sequences: list, str, tuple, bytes.

Each element in the sequence is addressed with an index.

Counting starting from 0

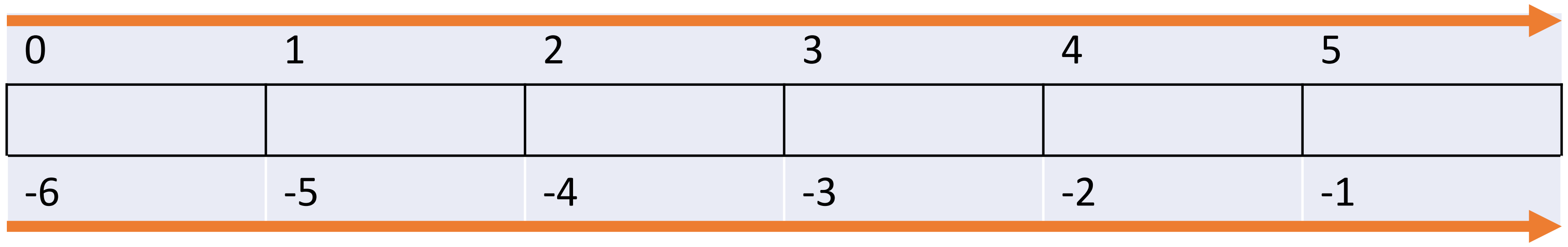
| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | |
| -6 | -5 | -4 | -3 | -2 | -1 |

## Python Sequences and Indexing

Built-in sequences: list, str, tuple, bytes.

Each element in the sequence is addressed with an index.

Counting starting from 0



## Python Sequences and Slicing

Slicing: accessing part(s) of a sequence

*SEQ = a sequence, a,b are int*

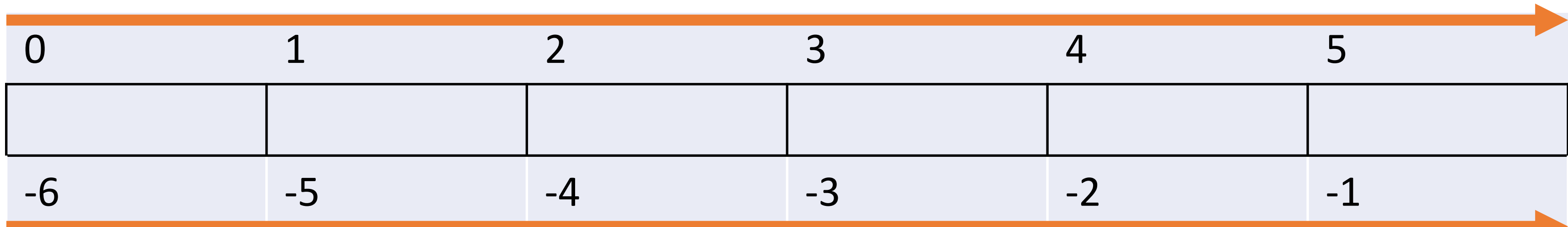
SEQ[a:b] (what if a > len(SEQ), b > len(SEQ), a and/or b negative

SEQ[:b], SEQ[a:],

SEQ[:]

SEQ[a:b:s]

SEQ[::-1]



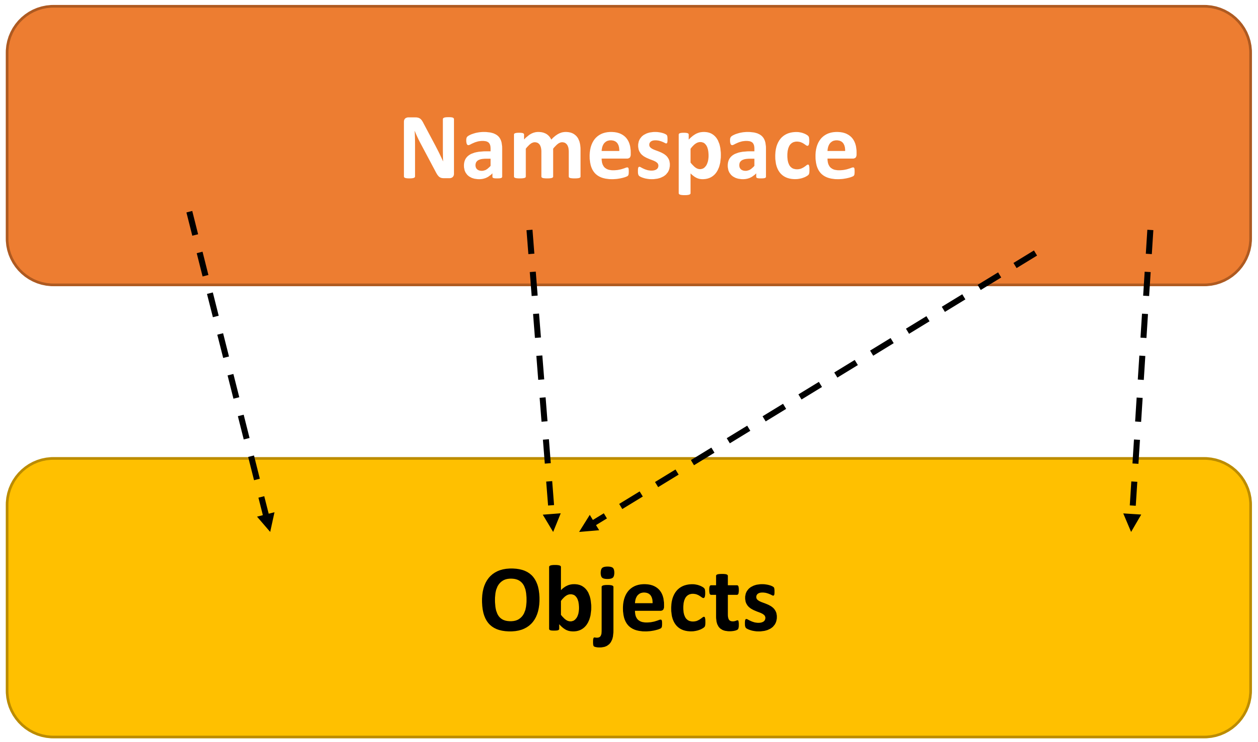
## Python Object & Reference

Mappings from namespace to objects,

one to one

or

one to many.

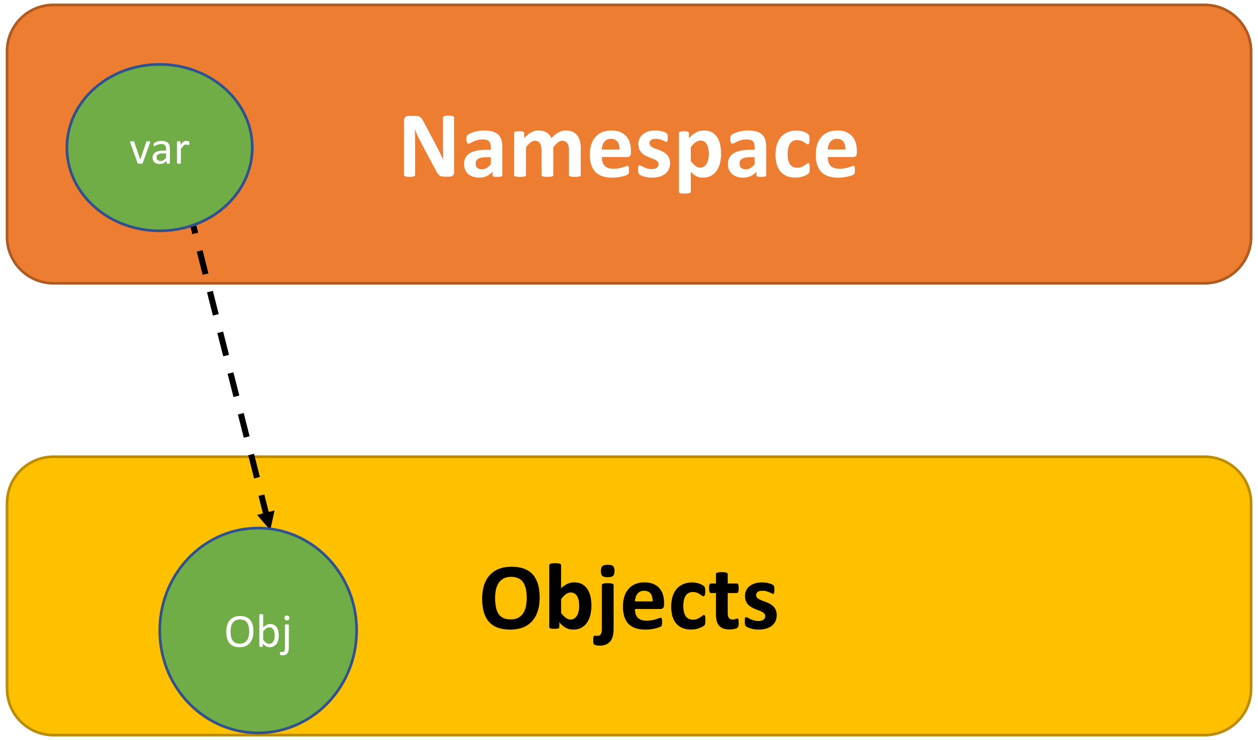


Every object have an Unique id.

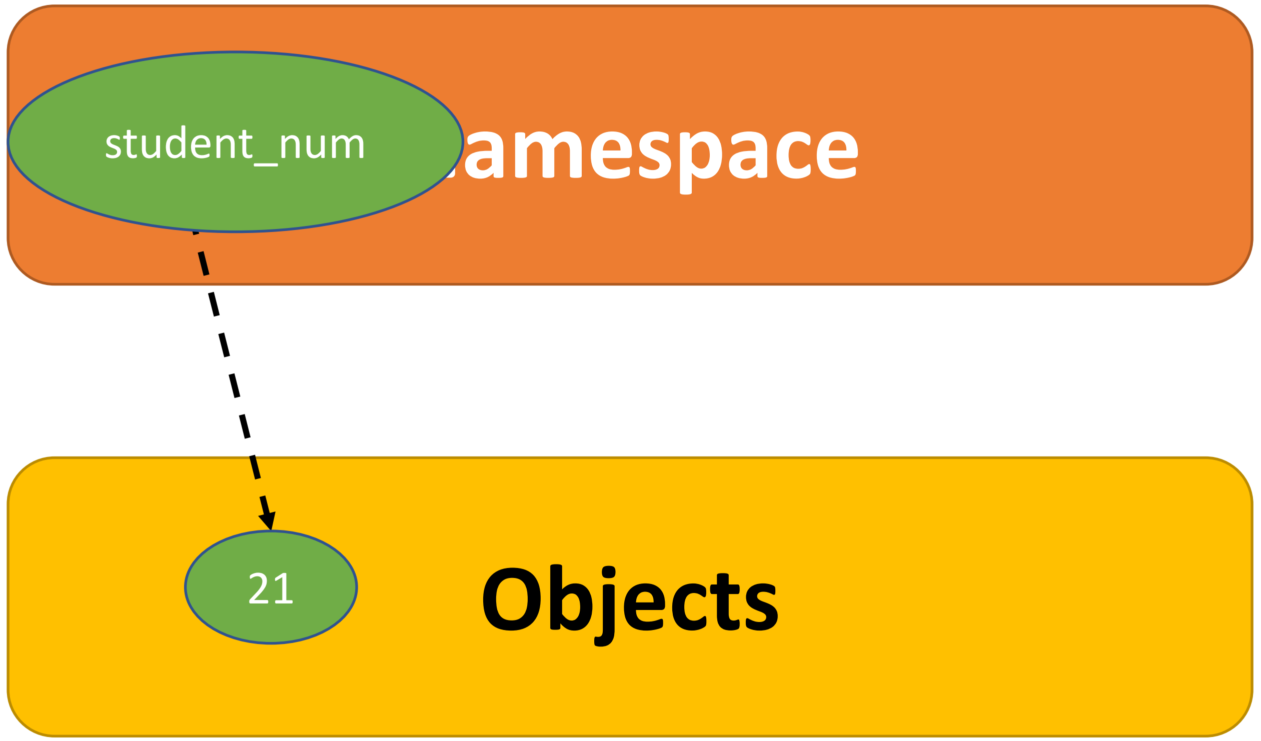
id() method

is operator

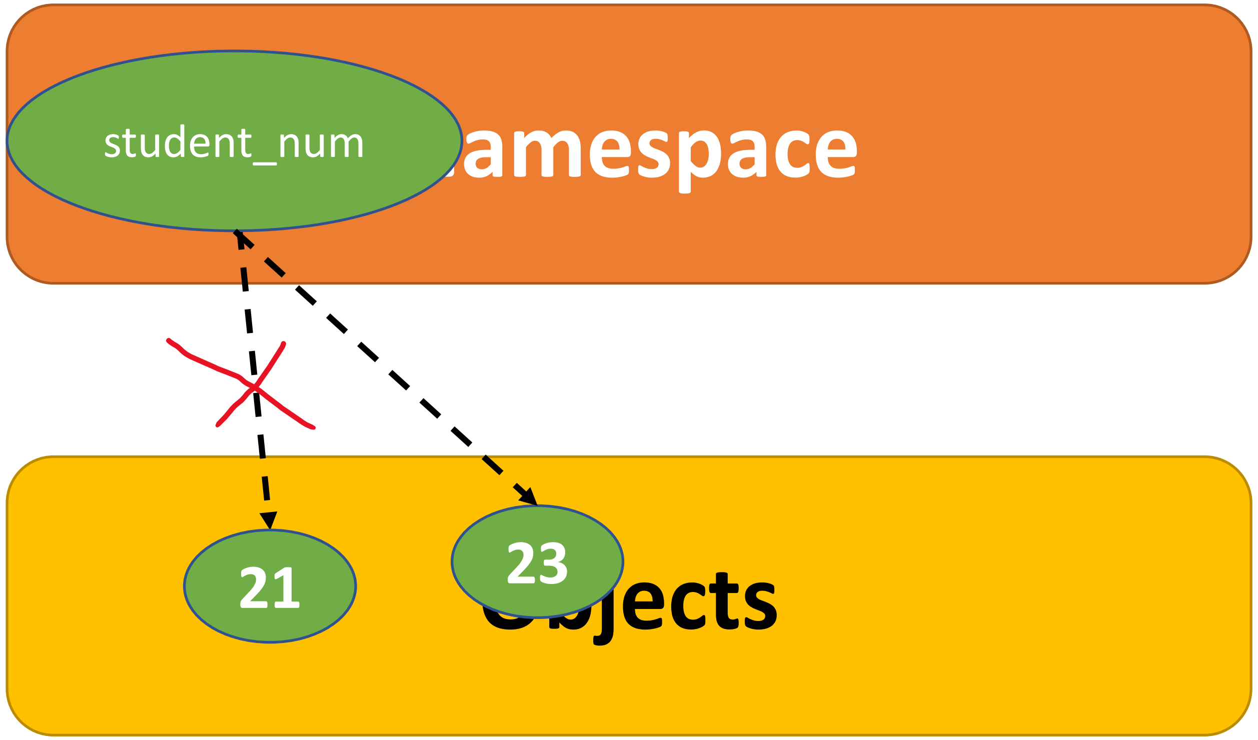
## Python Variables



## Python Variables



## Python Variables



## Python Object & Reference – Mutable and Immutable

Difference: can you change the object itself once it is created?

If yes:

        mutable

elif no:

        immutable

**lists, dictionaries, sets**

**Numbers, strings, bytes, tuples, Booleans**

## Python Memory Management

Automatic!

Reference counting.

Runtime keeps track of all references to an object.

When zero, unusable & can be deleted.

Additional memory and computation power required compared to manual memory management.

## Python Garbage Collection

By default, python uses CPython implementation.

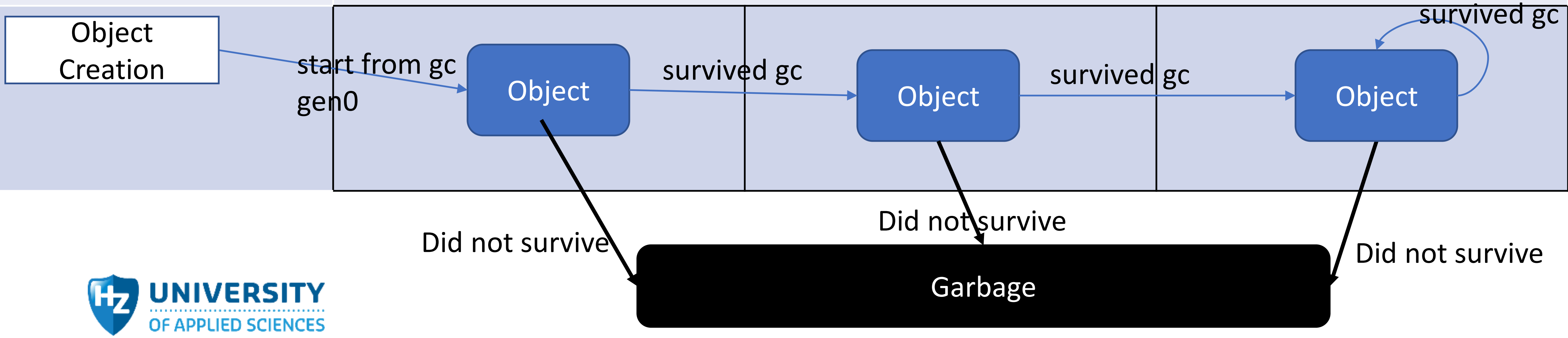
Uses:

- Reference counting
- Generational garbage collection
  - Cyclic garbage collector:
    - ✓ tracks all objects
    - ✓ have different generations (in total 3)
    - ✓ threshold in each generation (collect when it reaches)
    - ✓ does not work in real time (periodic runs)

## The Cyclic Garbage Collector

Many other conditions do exist!

| | Generation 0 | Generation 1 | Generation 2 |
|---|---|---|---|
| Frequency of garbage collection | High | | Low |
| Threshold | High<br>700 | 10 | Low<br>10 |



## Wrap-up

- ✓ Data types
- ✓ Numbers & strings
- ✓ Identifiers
- ✓ Operators
- ✓ Sequences, indexing and slicing
- ✓ Objects & references
- ✓ Memory Management & Garbage collection

## Wrap-up

✓ **Practice makes perfect.**