Brief introduction to Python

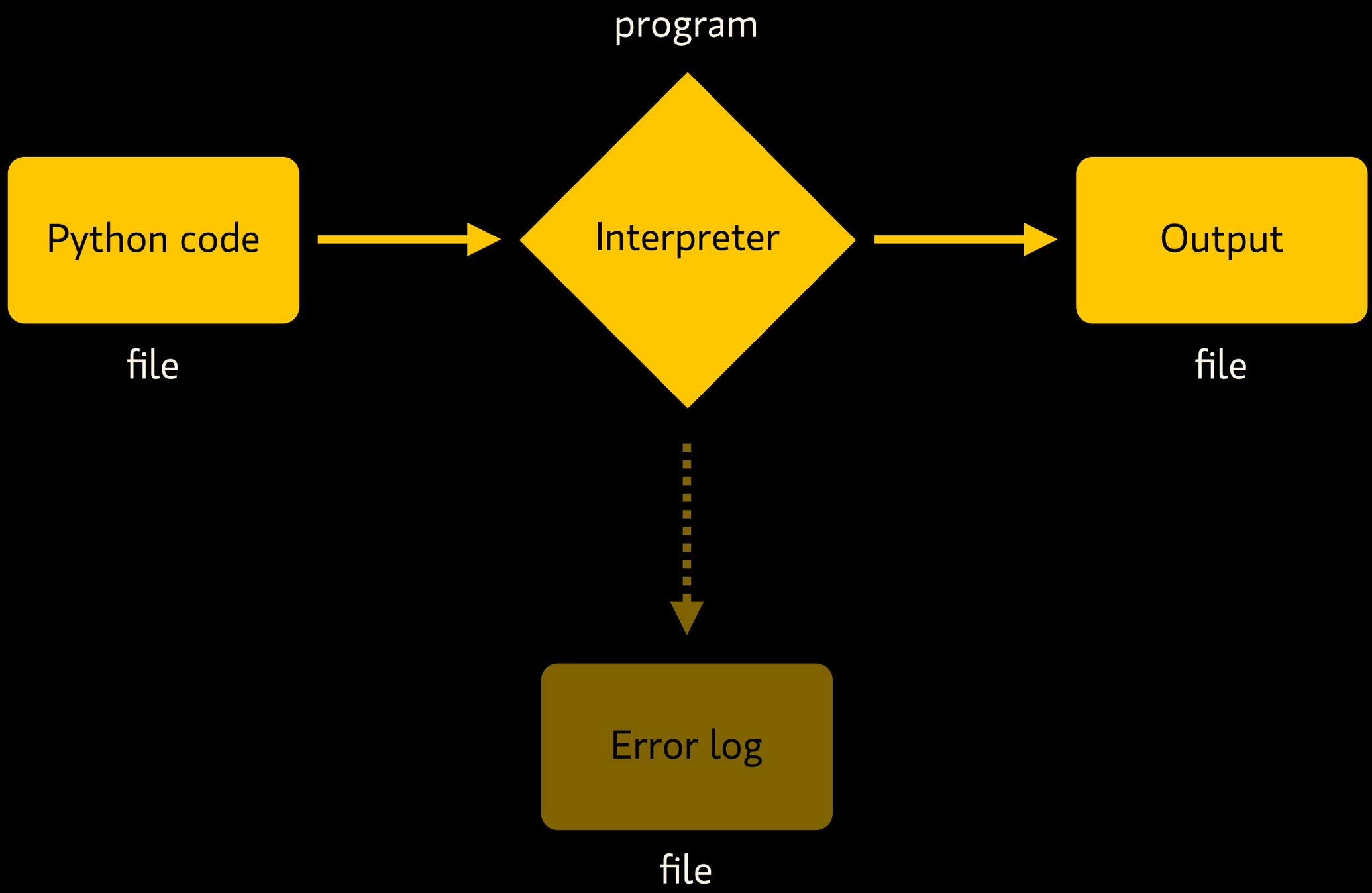
David Březina

Good and bad features

- “easy to use”
  - simplified syntax
  - indentation is part of the syntax
- object-oriented
- rather bad debugger
  - difficult to trace errors

Interpreter

- the interpreter is an application used to interpret any Python code
- the code is saved as a text file
  - .py extension is optional
- the output is a text file



Typical workflow of Python interpreter

Interpreting the code

- via Terminal (Shell, Command prompt)

  ```
  $ python script.py
  $ python script.py > output.txt
  ```

- using Python interactive prompt
- as a Shell script
- as a macro (e.g. in FontLab Studio)

Basic programming concepts

What programmers need to know

- programming concepts and strategies
- programming language
  - build-in types
  - build-in operators
  - build-in constructions
- extensions (libraries) to the programming language

Expression

- a formula made of operands and operators
- returns some result (numeric, logical, …)
- operands:
  - variables/attribute
  - constants/object
- operators:
  - build-in operators: +, -, <, >
  - functions/methods

Statement

- procedural unit of the script
- contains expressions
- "it says what the program should do"

Statement block

- set of statements grouped together
- grouped by indentation
  - same indent = same block

Object

- slot in the system memory
- contains value
- has some type

Type

- defines the object characteristics
- type is defined by:
  - allowed values
  - allowed operations
- types can be mutable or immutable
  - whether the object can be changed in place

Typing

- that is assigning type in general
- is dynamic in Python
- a programmer does not usually assign type
- type is connected with an object/value

2

value: 2, type: integer

3.14

value: 3.14, type: float

"a"

value: "a", type: string

Objects

Variable

- a reference to an object:
  - a unique name
  - link to an object which has:
    - value
    - type

Assignment

- use operator = to link variable with an object
- first assignment = initialising name, value & type

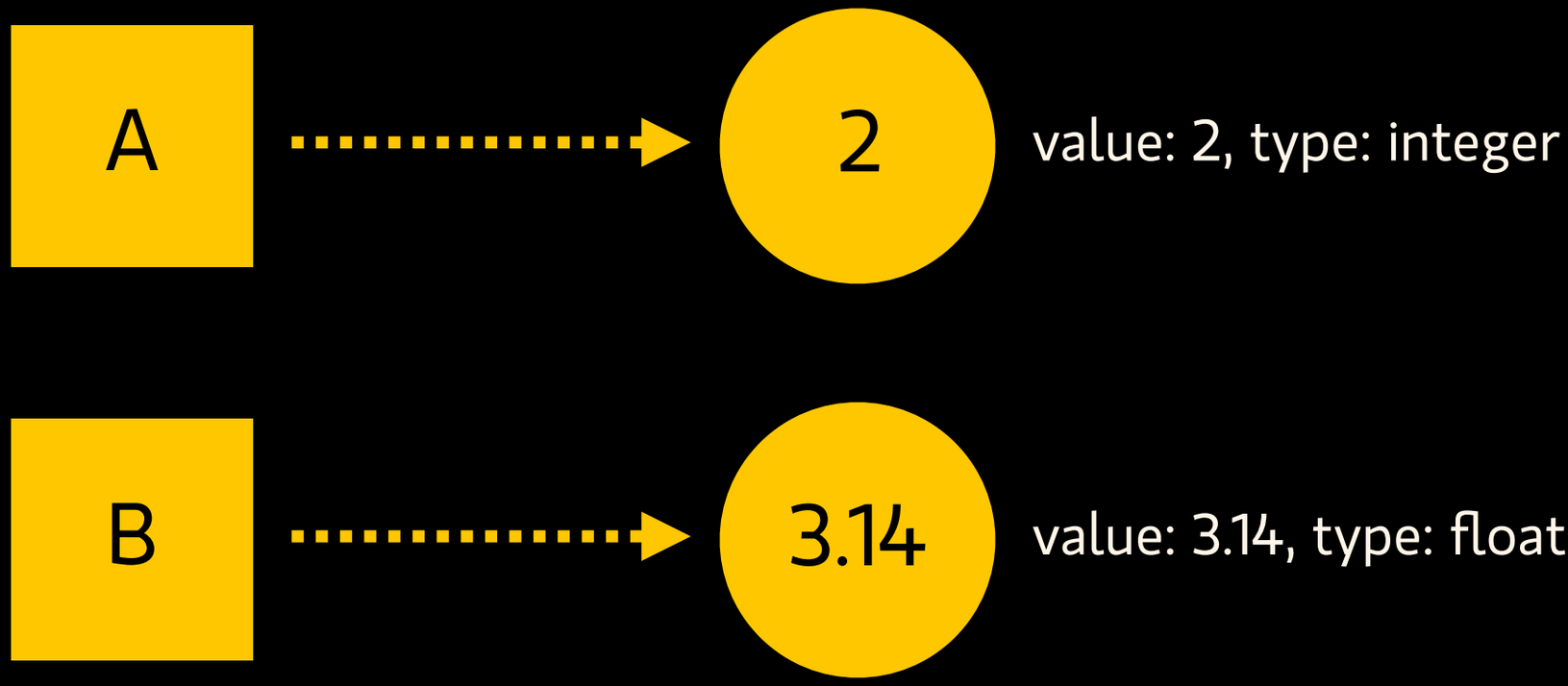
```
a = 2
```

- every other = change of value (and possibly type)

```
a = 3
```

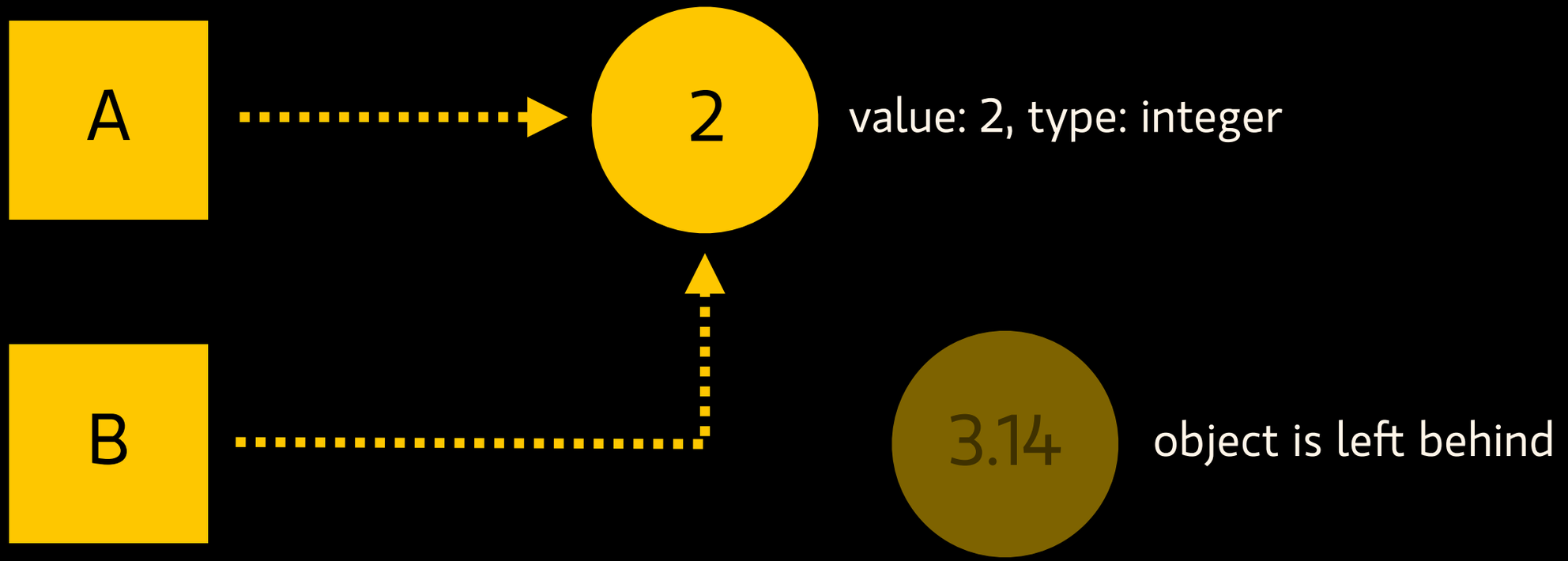
- assignments can “chain” in Python

```
a = b = 32
```



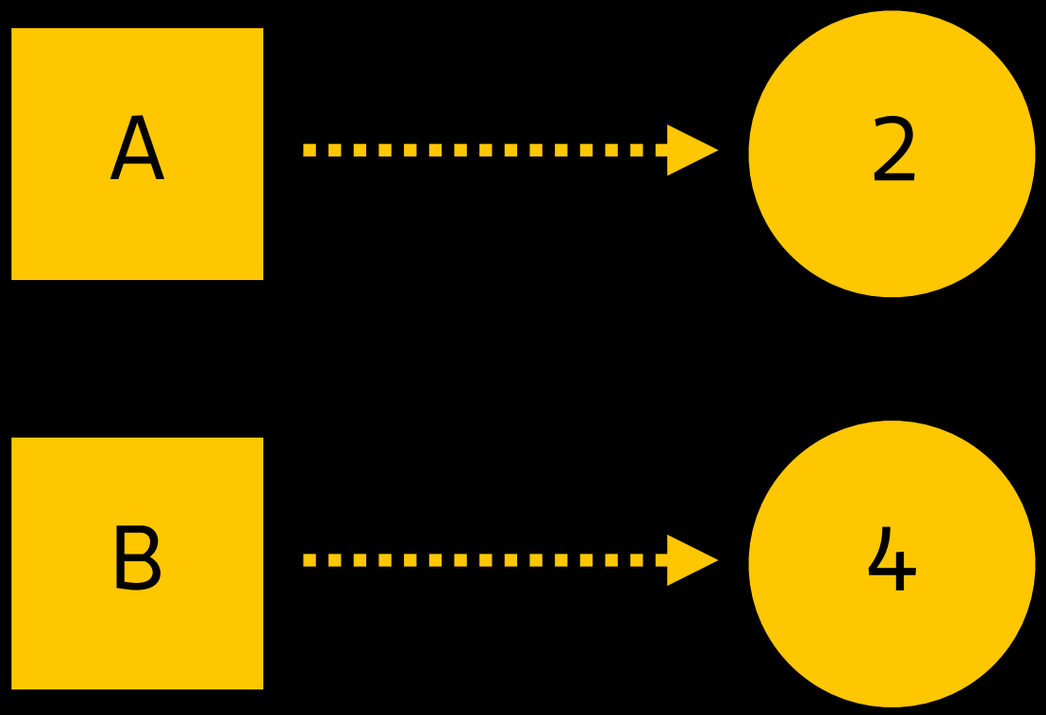
A = 2
B = 3.14

Variables & referencing
(variable is a reference to an object)



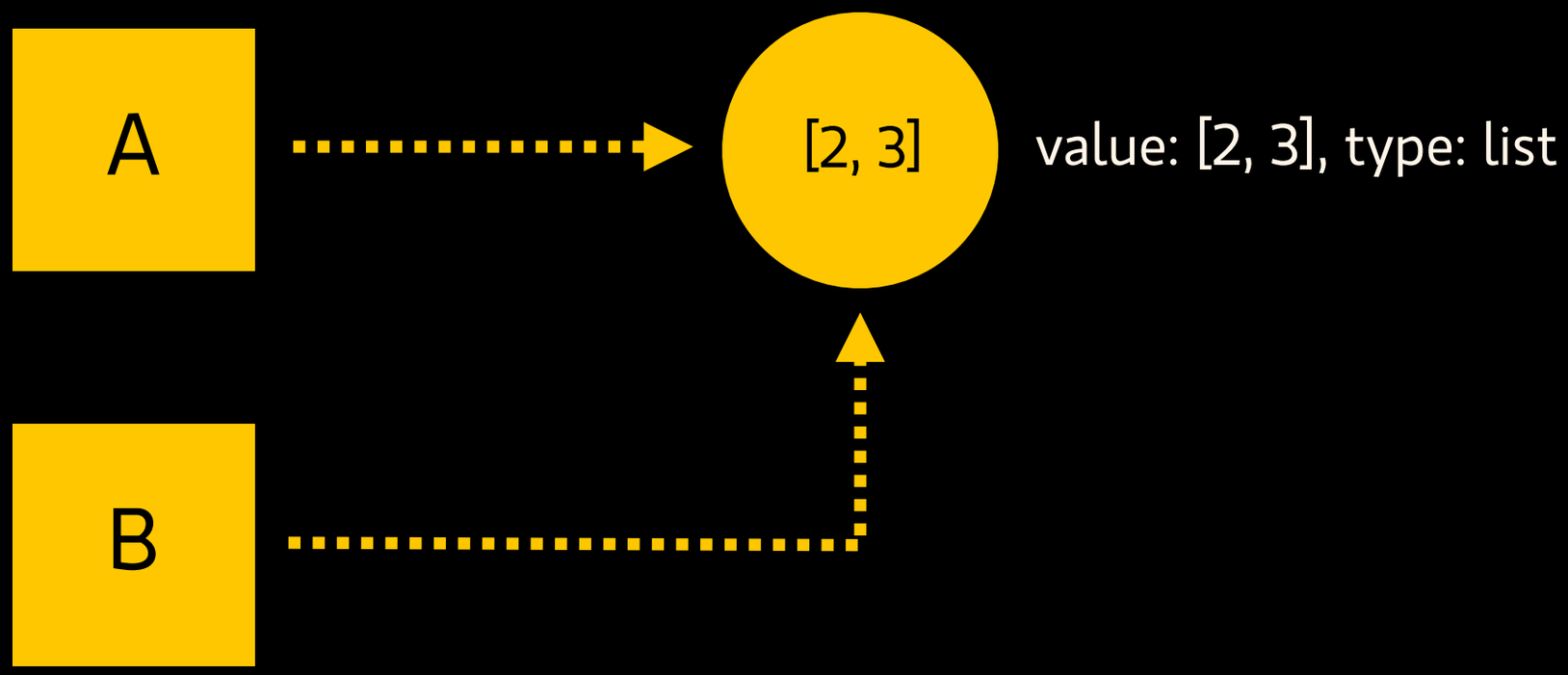
A = 2
B = 3.14
B = A

Variables & referencing
(two different variables link the same object)



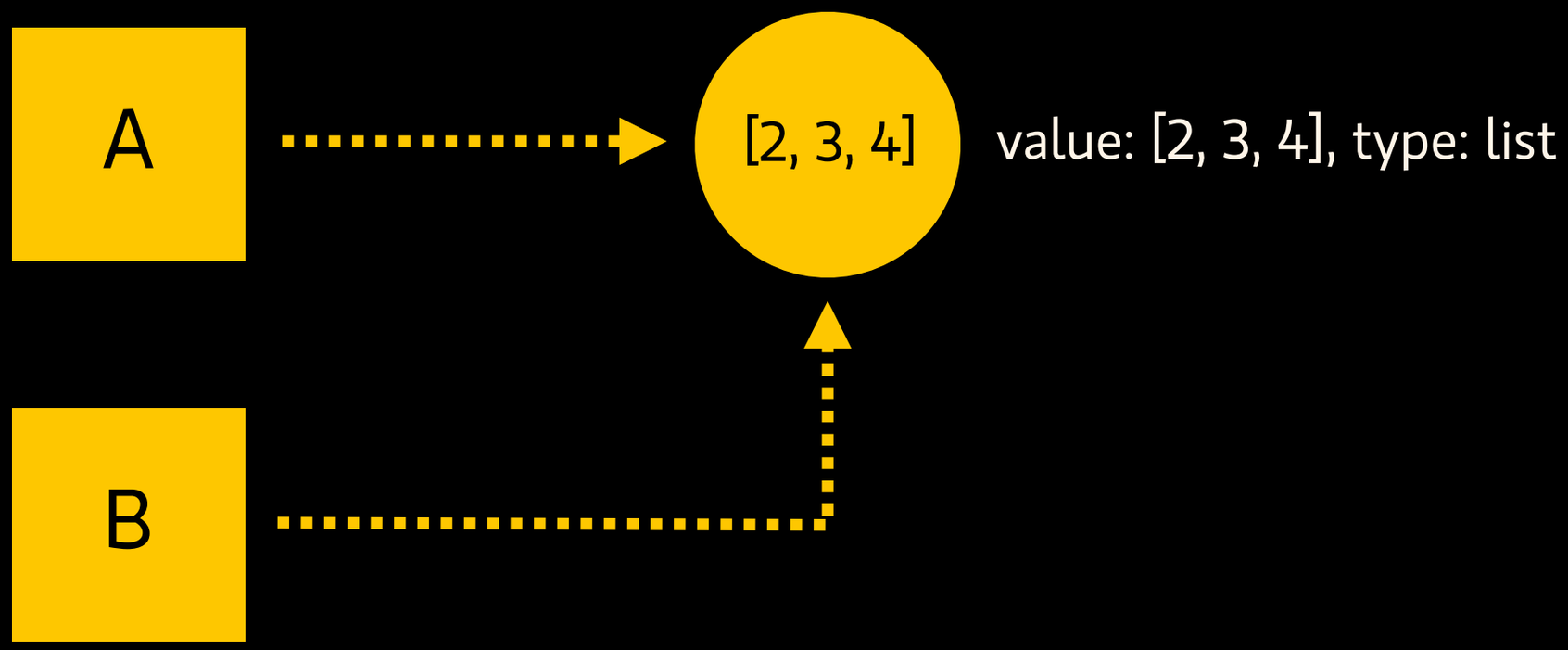
A = 2
B = 3.14
B = A
B = A + 2

Variables & referencing
(integers are immutable, link is released)



A = [2, 3]
B = A

Variables & referencing
(two variables refer to the same list)



A = [2, 3]
B = A
B.append(4)

Variables & referencing
(lists are mutable)

Comments

- everything after # is comment
- comments are not executed

```
# simple example to demonstrate
# basic Python syntax elements
# these are comments

a = 12      # assignment, intializing variable a
b = 3       # assignment, intializing variable b

if a > 20:        # conditional statement if
        # starting indented block
        a = 15      # assignment
        b = 4
        print "a is bigger than 20"   # print statement
        # end of indented block

else:
        # start of yet another indented block
        a = a + b + 1  # assigning result of an expression
        b = a - 5         # assigning result of an expression
        print "a is smaller than 20"
        # end of indented block

print a    # print statement
print b
```

Basic types & operations

Basic number types (int, float)

- integers (int)

  23, -47, 0

- float-decimal numbers (floats)

  3.14, -2.0

- typically immutable

Numbers: operations

- arithmetic operators:

  +, -, *, /, %, **

  +=, -=, *=, /=, %=, **=

- relational operators (result is boolean):

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

List (list)

- ordered sequence of items
- mutable
- items can be of any type
- even lists → nesting

```
emptyList = [ ]
l1 = ["a", 64, True, [3, 4, 5]]
l2 = [0, 1, 2, 3, 4, 5]
```

List: operations & methods

- sequence operations, indexing & slices

  +, *, a[*index*], a[*from:to*], len(*list*), in, del

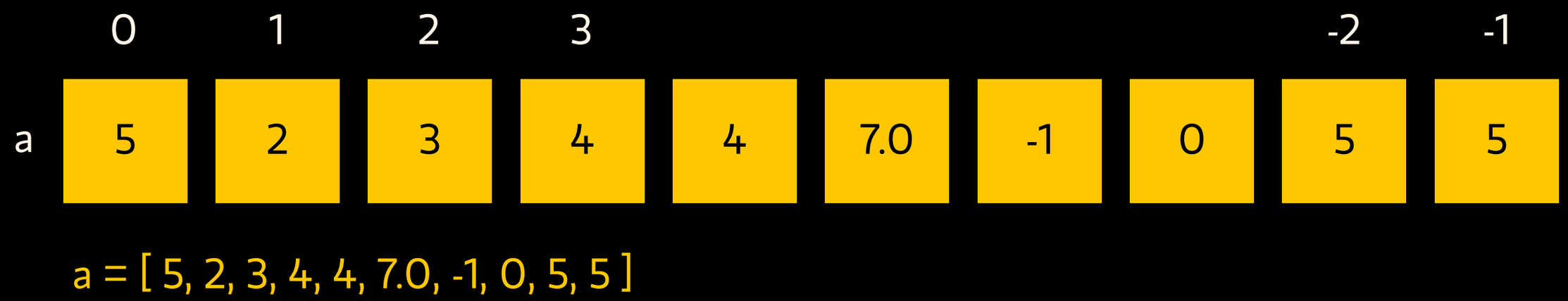
- type methods

  l.append()

  l.pop()

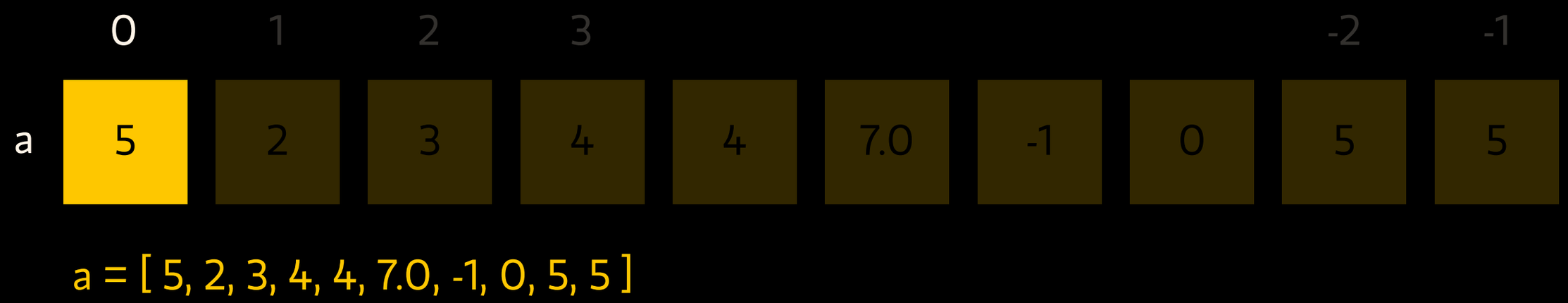
  l.sort()

  l.reverse()



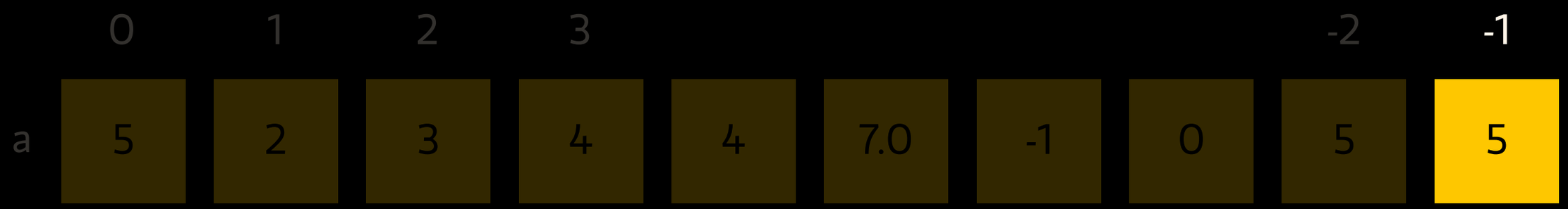
a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

Indexing & slicing
(zero-based, works same for strings)



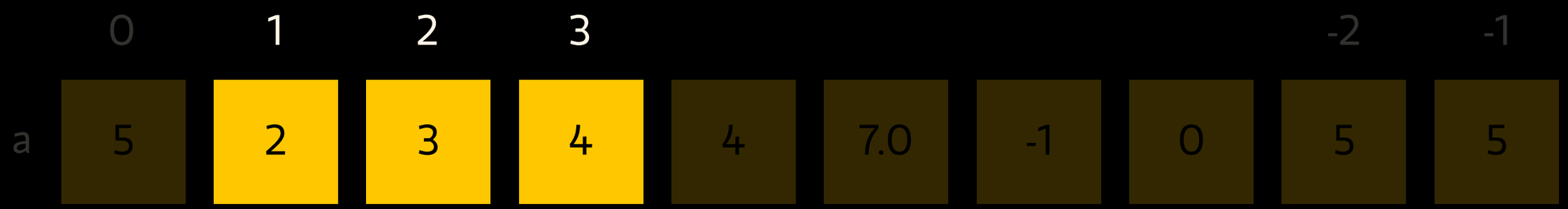
a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

Indexing & slicing
(a[0] is the first item)



a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

Indexing & slicing
(a[-1] is the last item)



a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

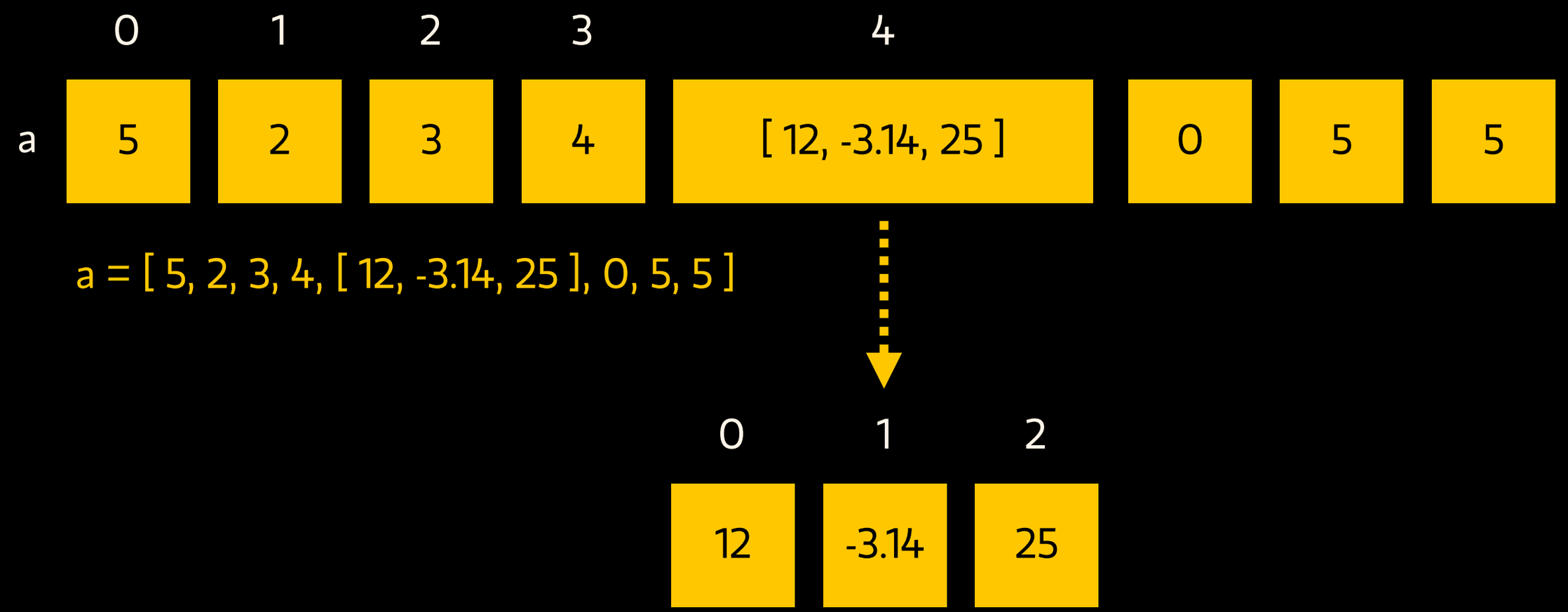
Indexing & slicing
(a[1:4] is a list of items from index 1 to 4 [excluded])

a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

Indexing & slicing
(a[1:] is a list of all items from index 1)

a = [5, 2, 3, 4, 4, 7.0, -1, 0, 5, 5]

Indexing & slicing
(a[:-2] is a list of all but last two items)



Nesting
(a[4] is a reference to another list, not a number)

String (str)

- ordered sequence (list) of characters
- immutable – you cannot change it
  - you can change it and rewrite it though

```
x = "this is string"
z = """string
running over multiple lines"""
emptyString = ""
```

String: operations

- sequence operations, indexing & slices

  +, *, a[*index*], a[*from:to*], len(*string*), in

- assignments

  +=, *=

- relational operators (result is boolean)

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

String: methods

str.upper()

str.lower()

str.capitalize()

str.find()

str.replace()

str.isalpha()

str.isdigit()

String: escape sequences

- \n for new line
- \" \' for quotes within " " resp. ' 'string

Tuple (tuple)

- immutable lists
- immutability is useful (esp. in functions):
  - the number of items does not change
  - the order does not change

```
emptyTuple = ( )
names = ("Patrick", "Alice", "Robin", "Dan")
anchor = ("top", 256, 1024)
```

Tuple: operations

- sequence operations & slices

  +, *, a[*index*], a[*from:to*], len(*tuple*), in

- assignments

  +=, *=

- relational operators (result is boolean)

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

Unpacking

- each item from a tuple on the right side is assigned to a variable on the left side
- the number of items on both sides has to be same

```
x, y, z = ("a", "b", "c")
st, ff, dd = "text", 2.5, 3256
```

Dictionary (dict)

- collection of doubles key:value
- mapping a series of keys with values, not sorted
- values of any type
- keys have to be immutable

```
D = { }
D = {"name":"David", "height": 182, "weight": 72}
```

Dictionary: operations

- access, assign & delete values

  d[*key*]

  d[*name*] = "Jakub"

  del d[*key*]

- type functions

  d.keys()

  d.values()

  d.items()

  has_key()

Boolean (bool)

- type to represent logical true or false
- often integer is used instead
- only two values:
  - True (non-zero value)
  - False (zero value)

Boolean: operations

- comparisons as explained before
- logical operations

  and, or, not

Boolean values of non-bool objects

- numbers are True if non-zero
- other are True if non-empty

None

- value-type placeholder
- no value, no type
- sometimes ussed as a return value of functions

- example:

```
a = None
```

Statements

Statements

- assignment
- function call
- print (should be a function)
- condition (if/elif/else)
- iteration
- loop
- import
- retyping

Text output (print)

```
print variable
print variable1, variable2
print "Decimal %d string %s &c." % (dec1, str1)
print
print "no line break after me",
```

Conditional statement (if)

- used to conditionally divert the code
- *condition* is any expression with boolean result

```
if condition:
    block processed
    when condition is True
```

Conditional statement (if/else)

if *condition*:

    block processed

    when *condition* is True

else:

    block processed

    when *condition* is False

Full conditional statement (if/elif/else)

```
if condition:
    processed when condition is True
elif anotherCondition:
    processed when anotherCondition is True
else:
    processed when neither of the previous
    conditions is True
```

Iteration (for/in)

- repeats following statement block for every item in the sequence (list, tuple, string)

```
for item in sequence:
    indented statement block
```

Iteration over list of numbers

- range(*from*, *to*) creates a list of integers

for *item* in range(*to*):

    indented statement block

for *item* in range(*from*, *to*):

    indented statement block

Iteration over list, with indexes

- enumeration(*list*) creates a list of tuples (index, item) and every tuple is unpacked

```
for index, item in enumeration(list):
    indented statement block
```

Iteration over dictionary

- the dictionary is converted to list of tuples (*key*, *value*) and every tuple is unpacked

```
for key, value in d.items():
    indented statement block
```

Conditional loop (while)

- repeats statement block while the *condition* is True

while *condition*:
indented
statement block

Importing module (import)

- imports module
- programmers have to use period notation to access items from the module (e.g. fl.font)

```
import fl
import robofab
```

Selective import (from/import)

- assigns objects/functions from the module to names in current name space
- programmer does not have to use the period notation to access objects from the module

```
from fl import font
from fl import *
```

Retyping

- use type name with ( )
- limited to meaningful conversions

```
str(11)
int("20")
tuple([1, 2, 3])
```

Functions & classes

Generic function definition

```
def functionName (argument1, arg2 = defValue2):

    statements included
    with the function

    return returnValue
```

Function calls

a = functionName (arg1, arg2)

functionName (arg1)

- argument 2 has default values defined, is optional
- you need to know number and type of arguments to call a function

Returning multiple values using tuple

```
def functionName (argument1, arg2 = defValue, ...):

    statements included
    with the function

    return returnValue1, returnValue2

a, b = functionName(arg1, arg2)  # function call
```

Introduction to name spaces/scopes

- variables defined locally in functions are not accessible anywhere else
- so-called global variables are accessible within functions
- good practise is to use function arguments instead of accessing global variables

Classes

- class is "object type" in object-oriented programming
- objects in o-o programming can include:
  - own variables → attributes
  - own functions → methods

- classes can customize already existing classes

Generic class definition

class *className* (*parentClass*):

    attribute1 = defaultValue1
    attribute2 = defaultValue2
    ...

    def *methodName1*(*arg1*, *arg2*, ...)
        method definition

    def *methodName2*(*arg1*, *arg2*, ...)
        method definition

Using an object

- creating an object
  o = *className*(*arg1*, *arg2*)
- using an attribute
  print o.*attribute1*
- creating an object
  o.*methodName1*(*a1*, *a2*, ...)

Macro debugging "tricks"

- comment-out pieces of code and add one by one back to see where is the problem
- print variables using print to see how they really look like and what they contain

Naming

- various approaches (e.g. CamelCase, under_scores)
- choose any which suits you and stick with it
- use descriptive, but not too long names, e.g.:
  - gl for glyph
  - font for font
  - i for iterated variable

Commenting

- do write comments!
- when you come back to the code to edit it you will not remember what it was supposed to do

Readable code

- readable = self-explanatory
- readable code is better than short code
- the more readable the code is the less you need to comment

Not covered

- other modules (sys, os, re)
- object-oriented programming, classes
- list comprehensions, matrixes
- regular expressions (module re)
- unicode strings
- files
- catching exceptions

Bibliography

The book used

Mark Lutz, Learning Python, O'Reilly, 2008.

Other sources

Python web: www.python.org

Python Tutorial: docs.python.org/tutorial/

Thinking in Python: www.mindview.net/Books/TIPython