AAE 875 – Fundamentals of Object Oriented Programming and Data Analytics

Cornelia Ilin, PhD

Department of Ag & Applied Economics

UW-Madison

Week 1 - Summer 2019

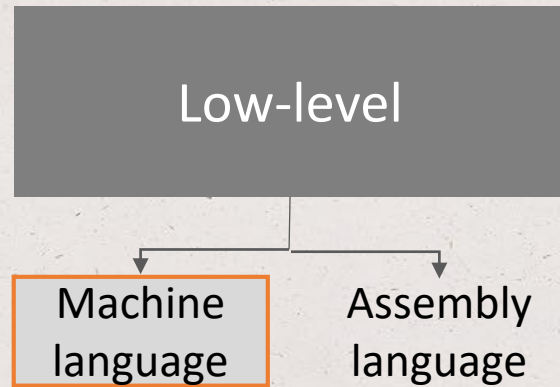# Programming languages - Types

Low-level

Created in 1940s

- 'low' because they are very close to how different hardware elements of a computer communicate with each other

- Require extensive knowledge of computer hardware and its configuration

Programming languages - Types

Low-level

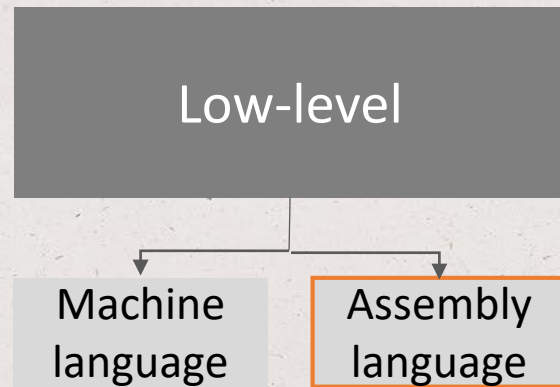Machine language   Assembly language

- The only language directly understood by a computer; does not need to be translated (by a compiler or interpreter – more on this later)

- All instructions use binary notations and are written as strings of 1s and 0s

011 1100001 001001 1100010

'machine code'

- However, binary notation is very difficult to understand -> develop *assembly language* to make machine language more readable by humans

Programming languages - Types

Low-level

Machine language   Assembly language

- Consists of a set of symbols and letters

- Requires an assembler to translate the assembly language to machine language

- A second generation because it no longer requires a set of 1s and 0s to write instructions, but terms like:

  Mul 97, #9, 98
  Add 96, #3, 92
  Div 92, #4, 97

```
        .globl  "_*add_forty_two<Int32>:Int32"
        .align  4, 0x90
"_*add_forty_two<Int32>:Int32":
        .cfi_startproc
        pushq %rbp
Ltmp1992:
        .cfi_def_cfa_offset 16
Ltmp1993:
        .cfi_offset %rbp, -16
        movq  %rsp, %rbp
Ltmp1994:
        .cfi_def_cfa_register %rbp
        addl  $42, %edi
        movl  %edi, %eax
        popq  %rbp
        retq
        .cfi_endproc
```

- Assemblers automatically translate assembly language instructions 'Mul 97, #9, 98', into **machine code** (011 1100001 001001 1100010).

- Easier than machine language but still difficult to understand -> develop high level languages

Programming languages - Types

Low-level

Machine language | Assembly language
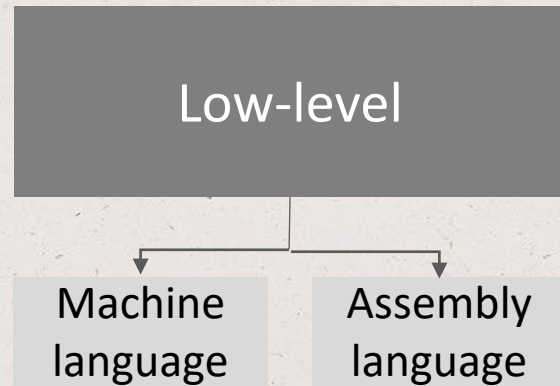
High-level

created in 1960s, 1970s

- Uses English and mathematical symbols in its instructions

- This is what most programmers use these days

- Examples: Fortran, Java, C++, Python

- More closer to the logic of human thinking:
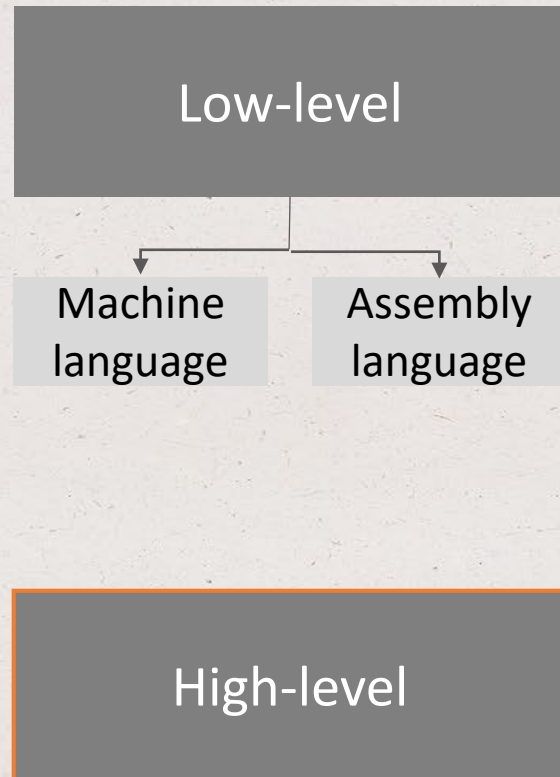
```
import random

numbers = range(1, 50)
chosen = []

while len(chosen) < 6:
    number = random.choice(numbers)
    numbers.remove(number)
    chosen.append(number)

chosen.sort()

print "This week's numbers are", chosen
print "The bonus ball is", random.choice(numbers)
```
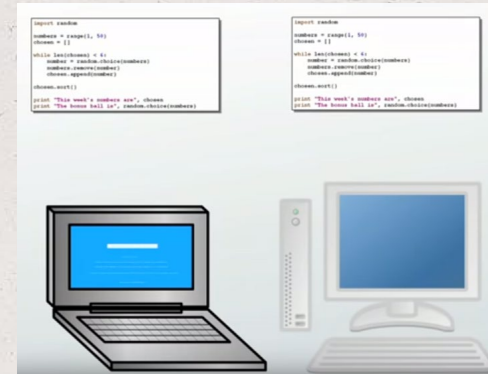
Programming languages - Types

Low-level

Machine language   Assembly language

High-level

- To learn to program in a high-level language you need to learn *commands*, *syntax*, and *logic*, which correspond closely to vocabulary and grammar

- The 'high-level' program is often portable, the same program can run on different hardware (not necessary OP)

- [Both machine language and assembly language are machine specific, and not portable (the machine code has to be modified to run in another computer)]

Programming languages - Types

Low-level

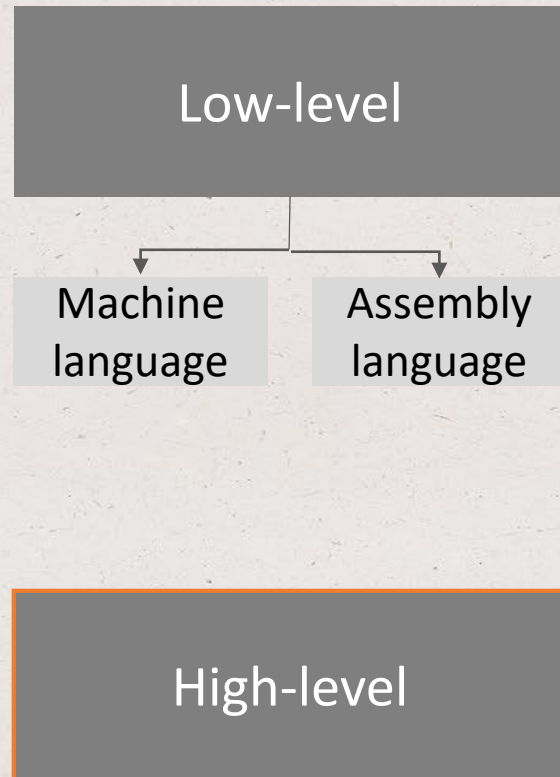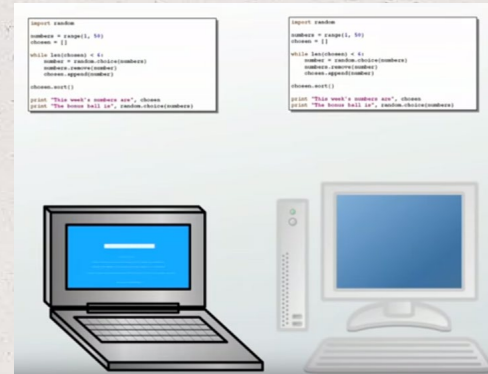Machine language   Assembly language

High-level

- To learn to program in a high-level language you need to learn *commands*, *syntax*, and *logic*, which correspond closely to vocabulary and grammar

- The 'high-level' program is often portable, the same program can run on different hardware (not necessary OP)
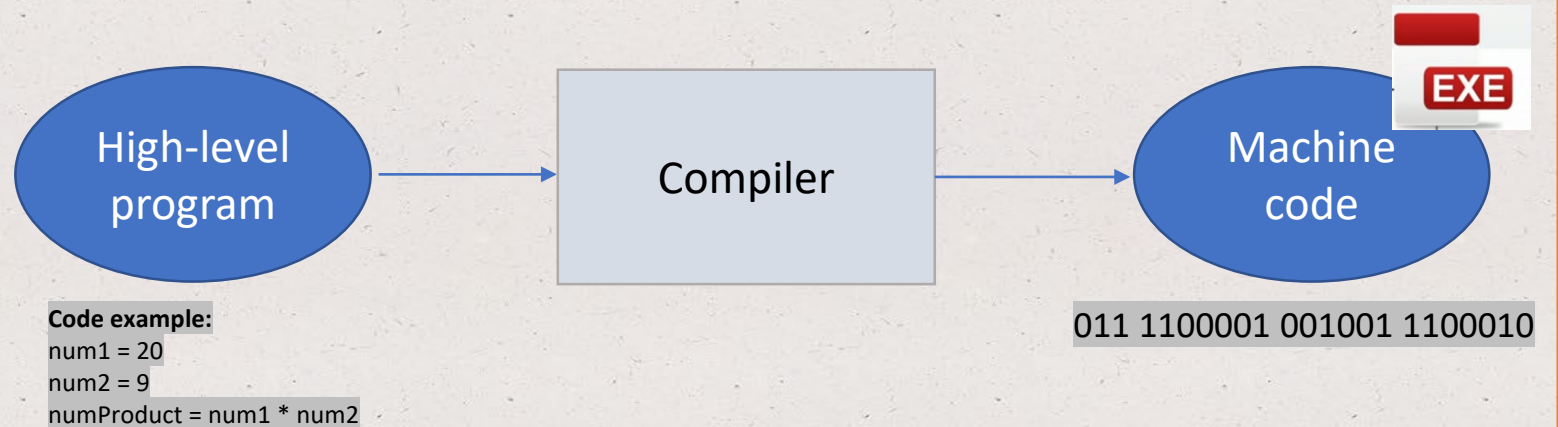

- The code cannot be directly understood by a computer, it needs to be *translated into* **machine code** using a **compiler** or **interpreter**

Programming languages - Types
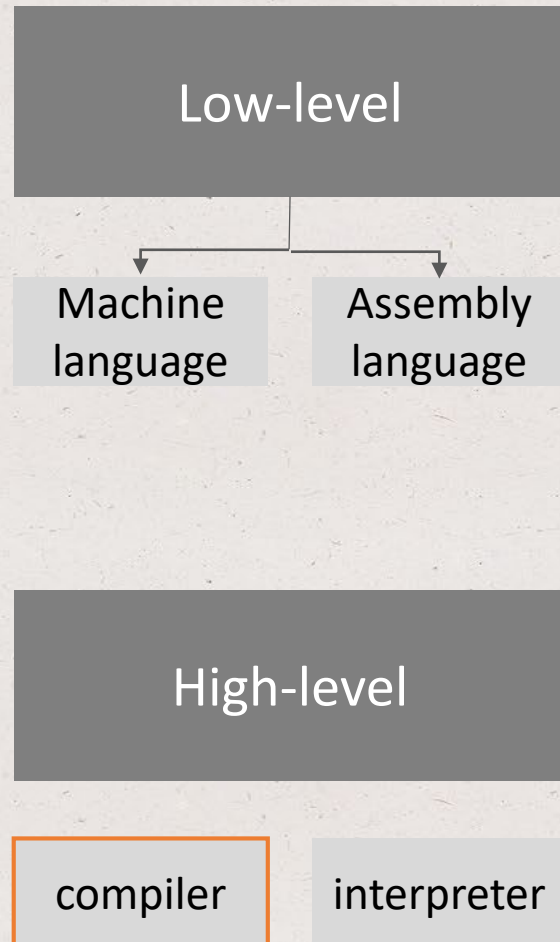
Low-level

Machine language   Assembly language

High-level

compiler   interpreter

- The compiler is used to translate a high-level program into machine code (the result is an .exe file)

High-level program → Compiler → Machine code

EXE

Code example:
num1 = 20
num2 = 9
numProduct = num1 * num2

011 1100001 001001 1100010

- Once you have an executable (.exe) file you can run the program over and over again without having to compile it again (e.g. you only need to translate (compile) a *movie* only once and can be used many times afterwards)

Programming languages - Types
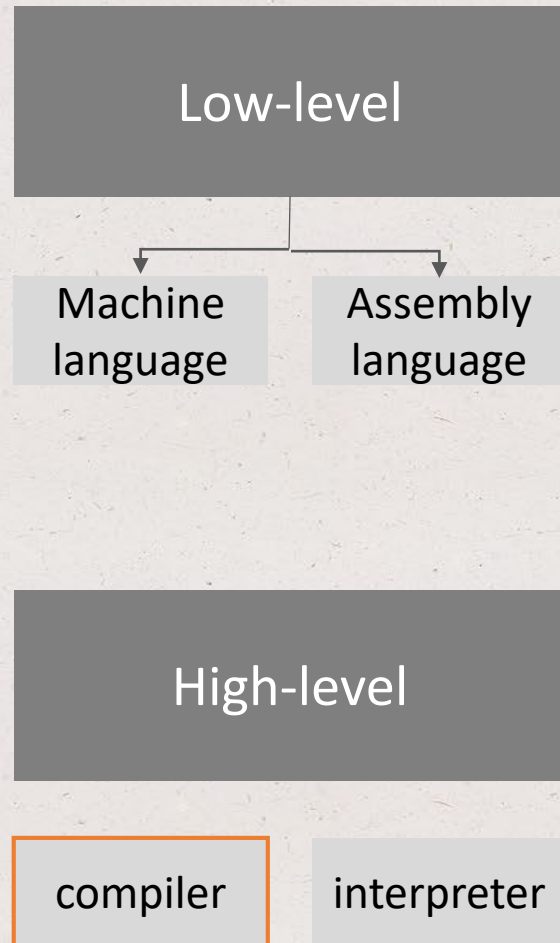
Low-level
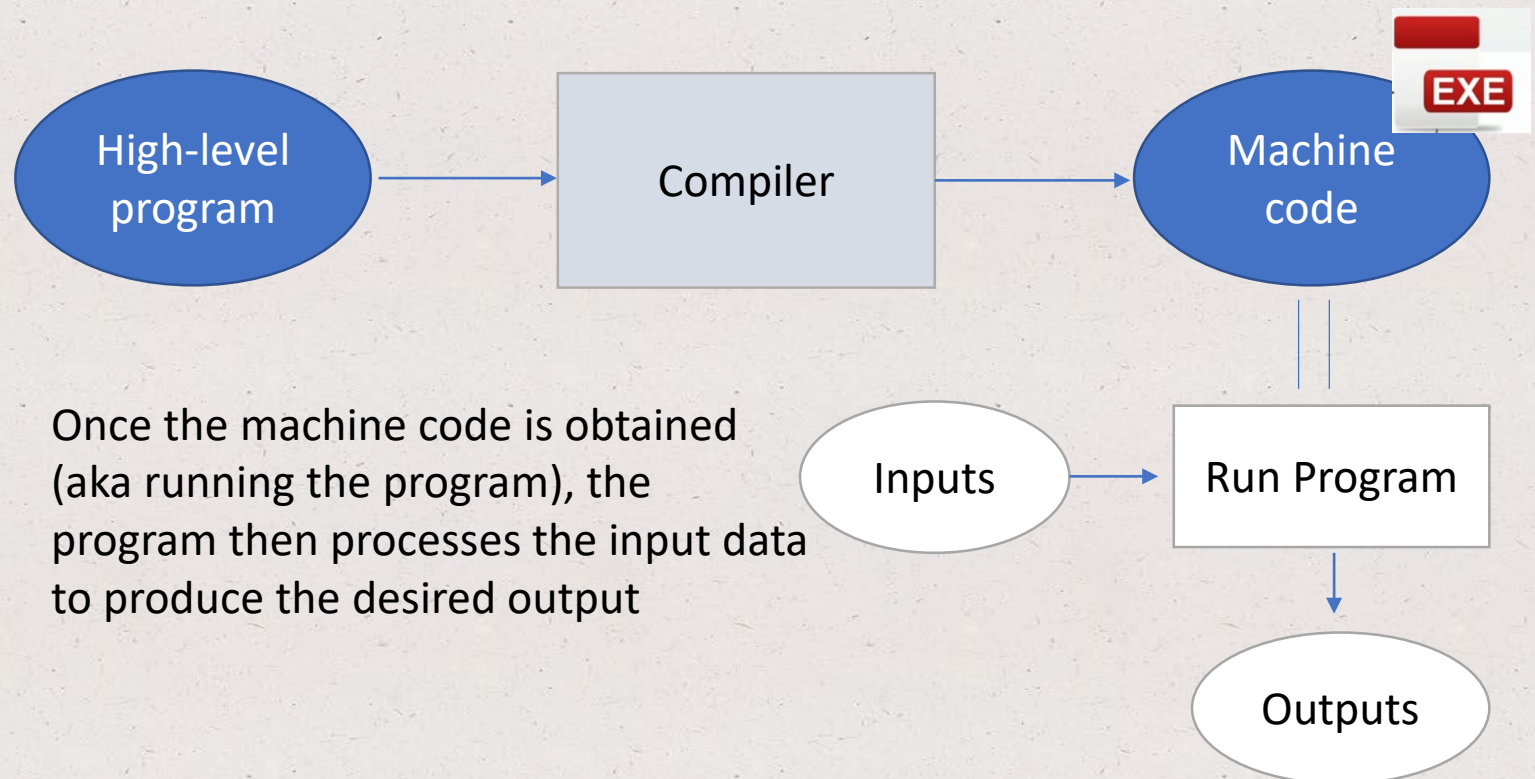
- Machine language
- Assembly language

High-level

- compiler
- interpreter

- The compiler is used to translate a high-level program into machine code (the result is an .exe file)

High-level program → Compiler → Machine code

Once the machine code is obtained (aka running the program), the program then processes the input data to produce the desired output

Inputs → Run Program → Outputs

Programming languages - Types

Low-level

Machine language   Assembly language

High-level

compiler   interpreter
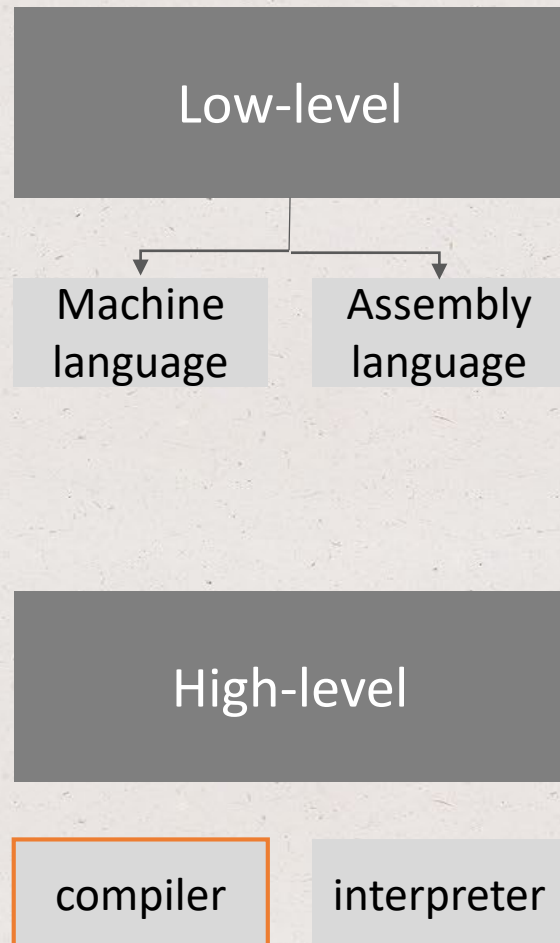
- **Advantages:**
  -> does not reveal the original source code (possible to distribute the program w/o revealing the inner workings; when you install a software application in your computer you typically install a compiled (.exe) version of the code
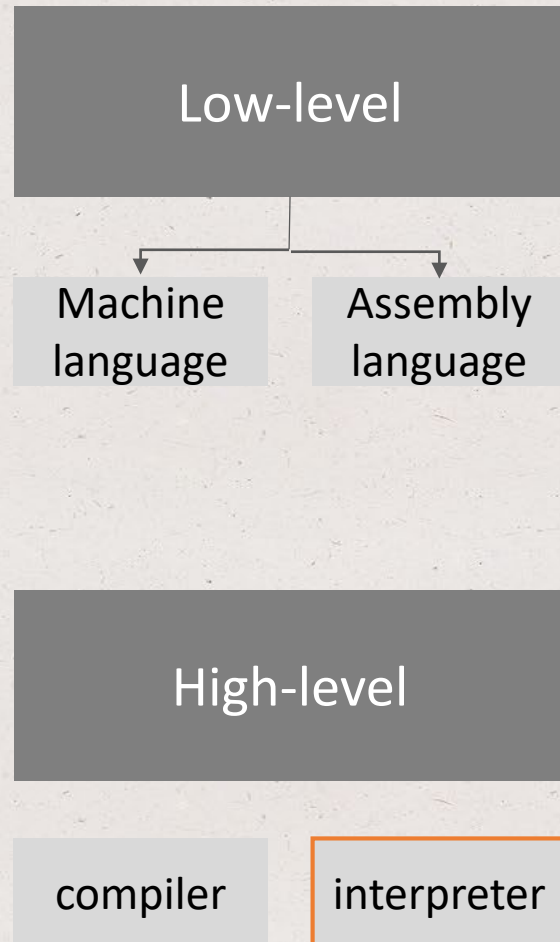


```
1100011
1001011
1110010
1101100
```
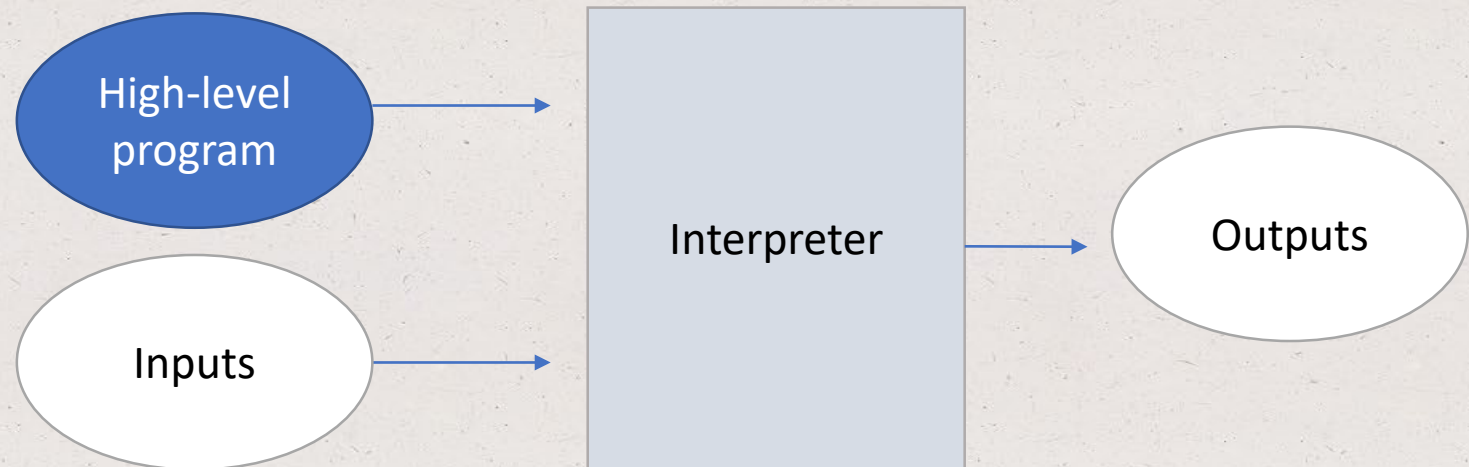
- **Disadvantages:**
  -> slower than interpreter languages

- **Examples:** C, C++, C#, Fortran, Java

Programming languages - Types

Low-level

Machine language   Assembly language

High-level

compiler   interpreter
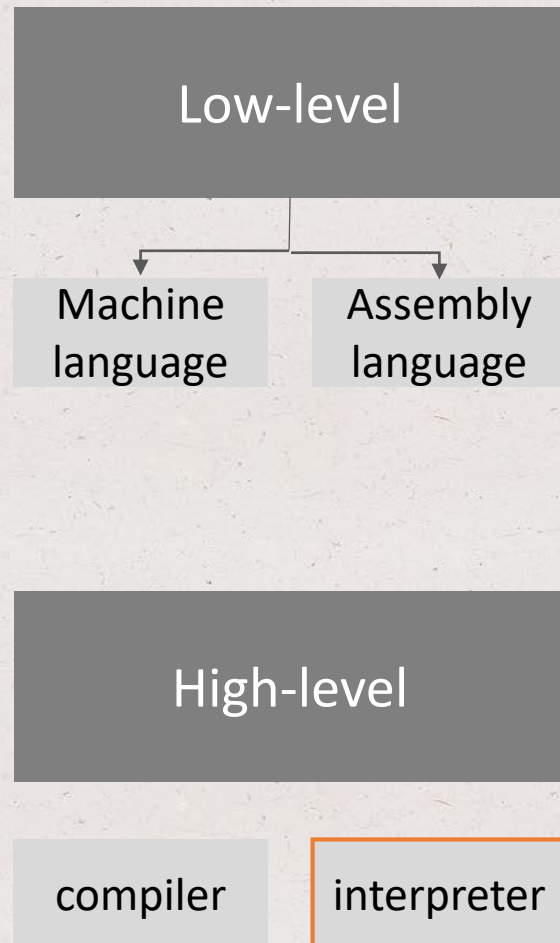
- An interpreter is a computer program that *simulates a computer that understands a high-level language*

- The interpreter translates the high-level program line by line during execution, which results in the desired output data

- The only result is the output data, there is **no compile code**

High-level program → Interpreter → Outputs

Inputs →

Programming languages - Types

Low-level

Machine language   Assembly language

High-level

compiler   interpreter

- When using an interpreter, every time you want to run the program you need to interpret the code again line by line (e.g. you translate (interpret) a *speech* only once and the results won't be used again)

High-level program →

Inputs →

Interpreter → Outputs

Programming languages - Types

Low-level

Machine language | Assembly language
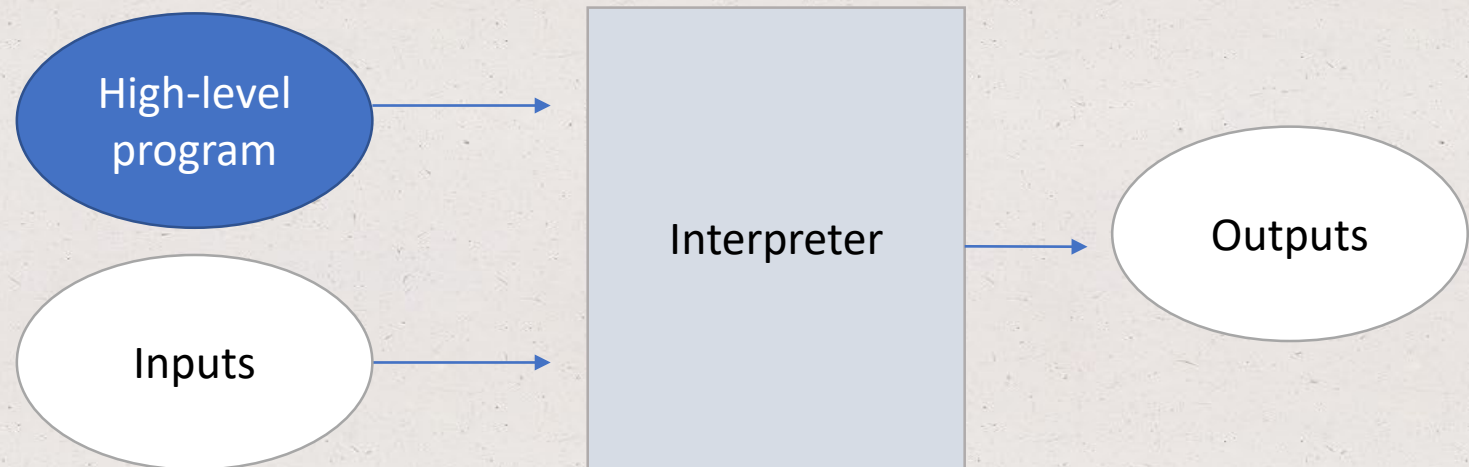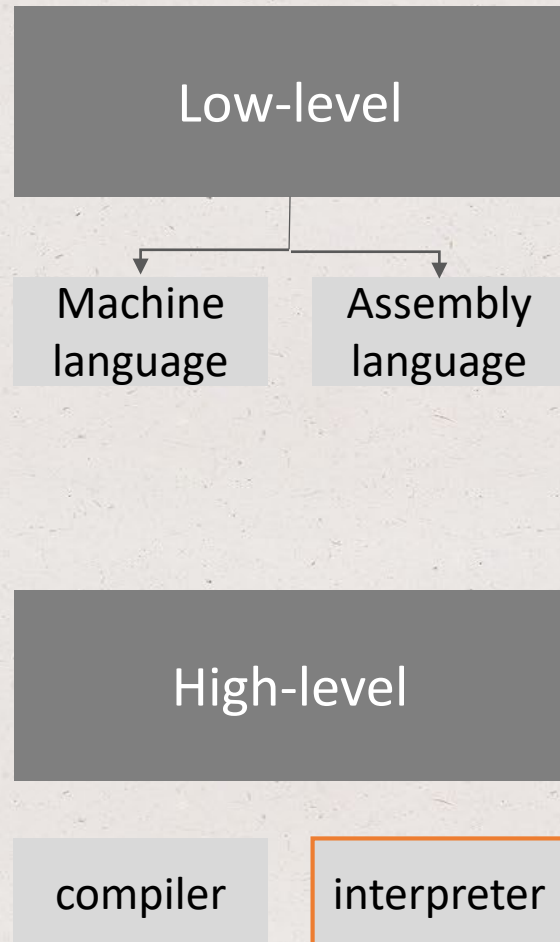
- **Advantages:**
  -> faster than compiler languages

- **Disadvantages:**
  -> not suitable for commercial software developments if you don't want to reveal the source code

- **Examples:** Python, Perl, Ruby

High-level

compiler | interpreter

Top Hat Question # 1

What type of programming language is Python?

(a) interpreted, low-level programming language
(b) compiled, low-level programming language
(c) interpreted, high-level programming language
(d) compiled, high-level programming language

Programming v. spoken languages

	Programming languages	Spoken languages
Syntax and structure	overlaps (e.g. print("hello world") in Java and Python)	overlaps (e.g. imprimer and imprimir are verbs for "print" in French and Spanich)
Natural lifespan	slowly die if no adoption (e.g. Algol, BCPL)	slowly die if no adoption (e.g. Latin, Aramaic)
Number of creators	can be created by one person (e.g. Python by Guido van Rossum)	multiple persons
Language	almost all (high-level) programming languages are written in English	of course, only English

How many programming languages?

How many programming languages?

- Wikipedia: over 700 'notable languages' in existence, both those in current use and historical ones

- https://en.wikipedia.org/wiki/List_of_programming_languages

Python – language history

- An interpreted, high-level programming language

- Invented in The Netherlands by Guido van Rossum

- Conceived in the late 1980s as a successor to the ABC language

- First released in 1990 (Python 2.0)

- Major update in 2008 (**Python 3.0**, not backward compatible,
  highly preferred)

Guido van Rossum

Guido van Rossum at the Dropbox headquarters in 2014

Born	31 January 1956 (age 63)[1] Haarlem, Netherlands[2][3]
Residence	Belmont, California, U.S.
Nationality	Dutch
Alma mater	University of Amsterdam
Occupation	Computer programmer, author
Employer	Dropbox[4]
Known for	Creating the Python programming language
Spouse(s)	Kim Knapp (m. 2000)
Children	1[5]
Awards	Award for the Advancement of Free Software (2001)
Website	gvanrossum.github.io

Python – language history

- Its philosophy: *code readability* and significant *whitespace*

- Python interpreters are available in many operating systems

- Guido van Rossum is fan of 'Monty Python's Flying Circus', a famous TV show in The Netherlands

- Named after Monty Payton

- **Open-sourced** from the beginning

Guido van Rossum

Guido van Rossum at the Dropbox headquarters in 2014

Born	31 January 1956 (age 63)[1] Haarlem, Netherlands[2][3]
Residence	Belmont, California, U.S.
Nationality	Dutch
Alma mater	University of Amsterdam
Occupation	Computer programmer, author
Employer	Dropbox[4]
Known for	Creating the Python programming language
Spouse(s)	Kim Knapp (m. 2000)
Children	1[5]
Awards	Award for the Advancement of Free Software (2001)
Website	gvanrossum.github.io

Chapter 1: Introduction

- Programming in general
- Development environment
- Basic Input and output
- Errors
- Computer tour

Programming in general

- A computer program = instructions (executed one at a time)

- Examples of basic instructions: **input** (   ), **process** (   ), **output** (   )

Programming in general

- A computer program = instructions (executed one at a time)

- Examples of basic instructions: **input** (x = 5, y = 6), **process** (x + y = 11), **output** (e.g. to a file)

Programming in general

- A computer program = instructions (executed one at a time)

- Examples of basic instructions: **input** (x = 5, y = 6), **process** (x + y = 11), **output** (e.g. to a file)

- A computer program is like a recipe (really, no difference!)

Humus and Tomato Pasta

1 tbsp pf olive oil

1 tsp of whole cumin seeds

1 large chopped onion

400 g chopped tomatoes

200 g humus

150 g pasta

1. Add the pasta to a large pan of boiling water. Simmer for 10 minutes

2. Fry the cumin in the olive oil for a few minutes. Add the onions and fry gently

3. Stir in the tomatoes and the humus and leave to simmer for 5 minutes

4. Drain the pasta and serve

Human

Humus and Tomato Pasta

1 tbsp pf olive oil

1 tsp of whole cumin seeds

1 large chopped onion

400 g chopped tomatoes

200 g humus

150 g pasta

1. Add the pasta to a large pan of boiling water. Simmer for 10 minutes

2. Fry the cumin in the olive oil for a few minutes. Add the onions and fry gently

3. Stir in the tomatoes and the humus and leave to simmer for 5 minutes

4. Drain the pasta and serve

Computer program

Humus and Tomato Pasta
/* List of inputs */
tbsp_olive_oil = 1
chop_onion = 1
gr_chop_tomatoes = 400
gr_humus = 200
gr_pasta = 150
boiled_water = 500 ml
tsp_cumin_seeds = 1

Human

Humus and Tomato Pasta

1 tbsp pf olive oil

1 tsp of whole cumin seeds

1 large chopped onion

400 g chopped tomatoes

200 g humus

150 g pasta

1. Add the pasta to a large pan of boiling water. Simmer for 10 minutes

2. Fry the cumin in the olive oil for a few minutes. Add the onions and fry gently

3. Stir in the tomatoes and the humus and leave to simmer for 5 minutes

4. Drain the pasta and serve

Computer program

Humus and Tomato Pasta
/* List of inputs */
tbsp_olive_oil = 1
chop_onion = 1
gr_chop_tomatoes = 400
gr_humus = 200
gr_pasta = 150
boiled_water = 500 ml
tsp_cumin_seeds = 1

/* Pasta processing */
pan1 = boiled_water + gr_pasta
Heat(pan1, low, 10 minutes)

/*Sauce processing */
pan2 = tbsp_olive_oil + tsp_cumin_seed
Heat(pan2, high, 2 minutes)
pan2 = pan2 + chop_onion
Heat(pan2, high, 2 minutes)
pan2 = pan2 + gr_chop_tomatoes + gr_humus
Heat(pan2, simmer, 5 min)

Human

Humus and Tomato Pasta

1 tbsp pf olive oil

1 tsp of whole cumin seeds

1 large chopped onion

400 g chopped tomatoes

200 g humus

150 g pasta

1. Add the pasta to a large pan of boiling water. Simmer for 10 minutes

2. Fry the cumin in the olive oil for a few minutes. Add the onions and fry gently

3. Stir in the tomatoes and the humus and leave to simmer for 5 minutes

4. Drain the pasta and serve

Computer program

Humus and Tomato Pasta
```
/* List of inputs  */
tbsp_olive_oil = 1
chop_onion = 1
gr_chop_tomatoes = 400
gr_humus = 200
gr_pasta = 150
boiled_water = 500 ml
tsp_cumin_seeds = 1


/* Pasta processing */
pan1 = boiled_water + gr_pasta
Heat(pan1, low, 10 minutes)


/*Sauce processing */
pan2 = tbsp_olive_oil + tsp_cumin_seed
Heat(pan2, high, 2 minutes)
pan2 = pan2 + chop_onion
Heat(pan2, high, 2 minutes)
pan2 = pan2 + gr_chop_tomatoes + gr_humus
Heat(pan2, simmer, 5 min)


/* Output (put it in a plate)*/
Drain(pan1)
Plate(pan1, pan2)
```

Human	Computer program

Human

Humus and Tomato Pasta

1 tbsp pf olive oil

1 tsp of whole cumin seeds

1 large chopped onion

400 g ~~chopped~~ tomatoes

200 g humus

150 g pasta

1. Add the pasta to a large pan of boiling water. Simmer for 10 minutes

2. Fry the cumin in the olive oil for a few minutes. Add the onions and fry gently

3. Stir in the tomatoes and the humus and leave to simmer for 5 minutes

4. Drain the pasta and serve

Computer program

Humus and Tomato Pasta

```
/* List of inputs  */
tbsp_olive_oil = 1
chop_onion = 1
gr_ chop _tomatoes = 400
gr_humus = 200
gr_pasta = 150
boiled_water = 500 ml
tsp_cumin_seeds = 1


/* Pasta processing */
pan1 = boiled_water + gr_pasta
Heat(pan1, low, 10 minutes)


/*Sauce processing */
pan2 = tbsp_olive_oil + tsp_cumin_seed
Heat(pan2, high, 2 minutes)
pan2 = pan2 + chop_onion
Heat(pan2, high, 2 minutes)
pan2 = pan2 + gr_chop_tomatoes + gr_humus
Heat(pan2, simmer, 5 min)


/* Output (put it in a plate)*/
Drain(pan1)
Plate(pan1, pan2)
```

Programming in general

- Important to develop **Computational thinking**
  - What is the main objective?
  - What are the inputs, outputs, their relationship?
  - What are the different components of the solution?

- **Computational thinking:** pseudocode it! then code it!

Pseudocode it!

/* **Main objective** is to cook Humus and Tomato Pasta

The inputs needed include olive oil, chopped onion, chopped tomatoes, humus, pasta, water, and cumin seeds

The first output is cooked pasta obtained by adding pasta to a pan of boiling water

The second output is the pasta sauce obtained my mixing olive oil, cumin seeds, chopped onion, tomatoes, and humus in a frying pan

The components of the solution are the first output and second output

The final output is the Humus and Tomato Pasta dish */

Code it!

Humus and Tomato Pasta
/* List of inputs */
tbsp_olive_oil = 1
chop_onion = 1
gr_chop_tomatoes = 400
gr_humus = 200
gr_pasta = 150
boiled_water = 500 ml
tsp_cumin_seeds = 1

/* Pasta processing */
pan1 = boiled_water + gr_pasta
Heat(pan1, low, 10 minutes)

/*Sauce processing */
pan2 = tbsp_olive_oil + tsp_cumin_seed
Heat(pan2, high, 2 minutes)
pan2 = pan2 + chop_onion
Heat(pan2, high, 2 minutes)
pan2 = pan2 + gr_chop_tomatoes + gr_humus
Heat(pan2, simmer, 5 min)

/* Output (put it in a plate)*/
Drain(pan1)
Plate(pan1, pan2)

Programming in general

- **Explain** your code: Summarize and provide a high-level explanation for what your code does in plain English! Python uses the pound sign (**#**) for comments

- **Trace** your code: Run the code as the computer does
  - Take a piece of paper (seriously!)
  - Write down the variables and their values
  - Update the variables as they change as you mentally walk through the statements sequentially

Programming in general

- An awesome tool for tracing Python code:

http://www.pythontutor.com/visualize.html#mode=edit

Development environment

- After installing Python 3 you will need to install an **integrated development environment (IDE)**

- An IDE is a software application consisting of an *editor*, build automation tools (e.g. compiler, *interpreter*), and a *debugger*

- Our preferred IDE is Eclipse. Supports the PyDev interpreter. You will use PyDev in this class

Basic output

- One way to print output is to use the built-in function **print()**
- Each **print()** statement will **output** on a **new line**

```
print('I am a resident of WI.')
print('I study at UW')
```

```
I am a resident of WI.
I study at UW
```

- Text enclosed in '' or "" is a **string literal**
- Allowed text in string literals: any letters, numbers, spaces, and any symbols like @#$

Basic output

- One way to print output is to use the built-in function **print()**
- Each **print()** statement will **output** on a **new line**

```
print('I am a resident of WI.')
print('I study at UW')
```

```
I am a resident of WI.
I study at UW
```

- How can we move text to the next line without using multiple print() statements?

Basic output

- One way to print output is to use the built-in function **print()**
- Each **print()** statement will **output** on a **new line**

```
print('I am a resident of WI.')
print('I study at UW')
```

```
I am a resident of WI.
I study at UW
```

- How can we move text to the next line without using multiple print() statements?
- Solution: use the **newline** escape **character \n** (should be part of the string!)

```
print('I\nam\nsmart')
```

```
I
am
smart
```

Basic output

- One way to print output is to use the built-in function **print()**
- Each **print()** statement will **output** on a **new line**

```
print('I am a resident of WI.')
print('I study at UW')
```

```
I am a resident of WI.
I study at UW
```

- How can we move text to the next line without using multiple print() statements?
- Solution: use the **newline** escape **character \n** (should be part of the string!)

```
print('I \n am \n smart')
```

```
I
am
smart
```

Basic output

- To keep the output on the same line: specify *end = ' '* as an argument of the print() function

print('I am a resident of WI.', end = ' ')
print('I study at UW.')

I am a resident of WI. I study at UW.

notice the space

Basic output

- How to output a variable's value? Use print(*variable*) -> without quotes

```
my_age = 20
print(my_age, end = '')
print('\'s')
```

20's

no space

Basic output

- How to concatenate items within a statement?

```
my_age = 20
print(my_age)
print('I am a resident of WI.')
print('I am', my_age, 'years old.')
```

```
20
I am a resident of WI.
I am 20 years old.
```

A comma separates multiple items. A **comma adds a whitespace** on the **left** side

Basic output

- How to concatenate items within a statement?

```
my_age = 20
print(my_age)
print('I am a resident of WI.')
print('I am', my_age, 'years old.')
```

```
20
I am a resident of WI.
I am 20 years old.
```

A comma separates multiple items. A **comma adds a whitespace** on the **left** side

Basic output

- How to concatenate items within a statement?

```
my_age = 20
print(my_age)
print('I am a resident of WI.')
print('I am'  , my_age  ,  'years old.')
```

```
20
I am a resident of WI.
I am 20 years old.
```

A comma separates multiple items. A **comma adds a whitespace** on the **left** side

Basic output

- How to concatenate items within a statement?

```
myAge = 20
print(myAge)
print('I am a resident of WI.')
print('I am ' , myAge , 'years old.')
```

```
20
I am a resident of WI.
I am 20 years old.
```

A comma separates multiple items. A **comma adds a whitespace** on the **left** side

Top Hat Question # 2

What is the output of:

print('2 and 2 =', 4)

Top Hat Question # 3

What is the output of:

print('2 and 2 = ', 4)

Basic input

- You can read input using the built-in **input()** function

- Reading from the input() function **always results in a string**

- String v. Integer?
    - '375' is a string, aka a sequence of characters '3', '7', '5'
    - 375 is an integer, the number three-hundred seventy-five

```
number = input()
print('My lucky number is', number)
```

```
375
My lucky number is 375
```

Type is string, i.e. '375'

Basic input

- You can read input using the built-in **input()** function

- Reading from the input() function **always results in a string**

- String v. Integer?

    '375' is a string, aka a sequence of characters '3', '7', '5'

    375 is an integer, the number three-hundred seventy-five

```
number = int(input())
print('My lucky number is', number)
```

```
375
My lucky number is 375
```

Type is integer, i.e. 375

- Use the built-in **int()** function to convert string to integer

Basic input

- You can add a **prompt in the input()** function as well

```
number = int(input('Enter your lucky number: '))
print('My lucky number is', number)
```

```
Enter your lucky number: 375
My lucky number is 375
```

- You can **perform operations** inside the **print()** function

Type is integer, i.e. 375

```
a = int(input())
b = int(input())
print(a * b)
print(a + b)
```

```
a = 5
b = 5
25
10
```

Errors

- **SyntaxErrors** (occur before the program is run by the interpreter, prior to executing code)

```
day = 4
print('Today is July' 4)
```

```
File '<main.py>', line 2
    print(Today is July' 4)
                        ^

SyntaxError: invalid syntax
```

What is missing in the syntax?

- **RuntimeErrors** (occur when the program is run by the interpreter, during code execution)

  Types: IndentionError, ValueError, NameError, TypeError

- **LogicErrors** (do not stop the execution of the program but the code does not behave as intended)

```
print( 2 * 4)
```

```
print( 2 * 40)
```

Top Hat Question # 4

Do you get any error when running this code? If yes, what type of error?

```
status = 'sunny'
print('Today is' + status)
```

Chapter 2: Variables and expressions

- Variables
- Expressions
- Objects
- Modules

Variables

- In programming, a variable is a named **item** that **holds the value** of an expression.

var name

| hot_days |

hot_days = 5 ← data type is integer

- Variables are *not declared* in Python. Their *data type is inferred* based on the assigned value

- Thus, Python is a **dynamic typed language**

- The data type of a variable can change depending on the assigned value

Variables

- **Naming Rules:**
  - Start with a letter or underscore (_)
  - Subsequent characters can be letters, digits, or underscores (_)
  - Can be any length (but choose shorter and meaningful names)
  - Case-sensitive: days ≠ Days
  - Cannot be a keyword used by Python (e.g. print, and, while)
  - -> Good practice: use all lowercase letters and place underscores between words

Variables

- **Naming Rules:**
  - Start with a letter or underscore (_)
  - Subsequent characters can be letters, digits, or underscores (_)
  - Can be any length (but choose shorter and meaningful names)
  - Case-sensitive: days ≠ Days
  - Cannot be a keyword used by Python (e.g. print, and, while)
  - -> Good practice: use all lowercase letters and place underscores between words
- **Can assign multiple variables at once** (allows for different data types)

```
hot_days, month = 5, 'June'
```

```
hot_days = 5
month = 'June'
```

Variables

- **Naming Rules:**
  - Start with a letter or underscore (_)
  - Subsequent characters can be letters, digits, or underscores (_)
  - Can be any length (but choose shorter and meaningful names)
  - Case-sensitive: days ≠ Days
  - Cannot be a keyword used by Python (e.g. print, and, while)
  - -> Good practice: use all lowercase letters and place underscores between words
- **Can assign multiple variables at once** (allows for different data types)
- **Can swap variable values**

```
hot_days, month = 5, 'June'
```

```
hot_days = 5
month = 'June'
```

```
hot_days, month = month, hot_days
```

```
hot_days = 'June'
month = 5
```

Variables

- **Naming Rules:**
  - Start with a letter or underscore (_)
  - Subsequent characters can be letters, digits, or underscores (_)
  - Can be any length (but choose shorter and meaningful names)
  - Case-sensitive: days ≠ Days
  - Cannot be a keyword used by Python (e.g. print, and, while)

  -> Good practice: use all lowercase letters and place underscores between words

- **Can assign multiple variables at once** (allows for different data types)

- **Can swap variable values**

- **Can delete variables**

hot_days, month = 5, 'June'

hot_days = 5
month = 'June'

hot_days, month = month, hot_days

hot_days = 'June'
month = 5

del hot_days
del month

Top Hat Question # 5

- **Write code to swap the values of a and b**

a = 2
b = 3

→

a = 3
b = 2

Data types

```
              Primitive                                                    Non-primitive

Integer    Float    Boolean    String      List    Tuple    Array    Dictionary    Set    File
```
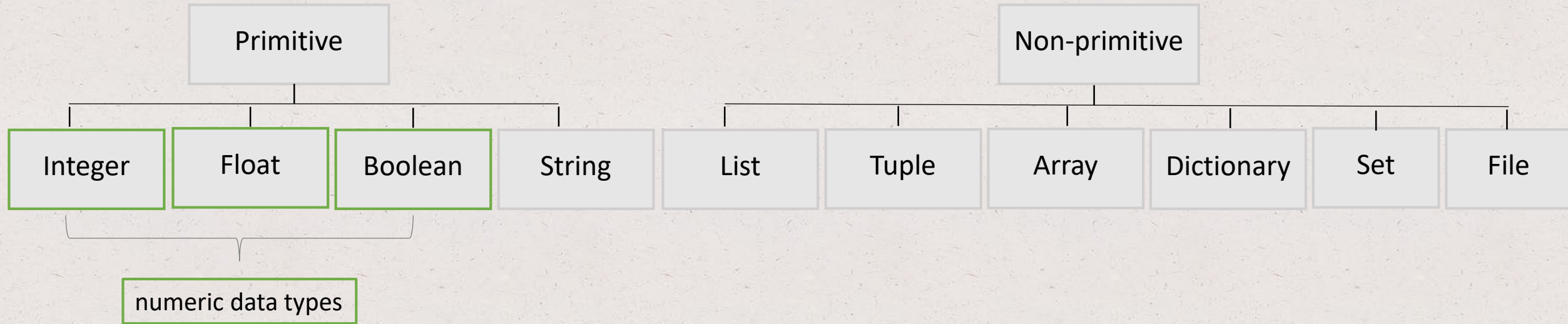
Data types

Numeric data types

- **Integer:** used to represent whole numbers from negative infinity to infinity, like 1, 2, 3 or -1, -2,...

- **Float:** used for rational point numbers, usually ending with a decimal figure, like 1.11 or 2.54

- Remember that in Python you don't need to explicitly state the variable type because Python is a dynamically typed language

- To find the class of a variable:

```
type(variable)
```

Numeric data types

- **Float:** the term 'float' because the decimal point can appear (float) anywhere

- Assigning a floating-point value outside of the allocated range results in an **OverflowError**

- To read a float type as an input:

  float(input('type a float number: ')

- **Literals:** represent a specific value assigned to a variable (e.g. integer literal, float literal)

Top Hat Question # 6

Which number is not a float?

(a) 1.0
(b) .55
(c) 2.3e2
(d) 4

Variable assignment

- The assignment operator (=) **assigns** a value to a variable

var_name = expression	a = b = c = 2 + 3

- The assignment operator (=) is right-to-left associative
  - Evaluate expression 2 + 3
  - Assign the value of expression to var_name

Top Hat Question # 7

What is the difference between an expression and an assignment?

Expressions – operators

- **Unary:**
  - -: negation
- **Binary:**
  - +: addition
  - -: subtraction
  - *: multiplication
  - /: division
  - **: exponent
  - %: reminder (modulo)
- **Compound:**
  - +=: add and assign (e.g. a += 3 is shorthand for a = a + 3)
  - Other variants: -=;   *=;   /=;   %=

Expressions – operator precedence

- An expression is evaluated using the order of standard mathematics

Operator/Convention	Description	Explanation
()	Items within parentheses are evaluated first.	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
unary -	- used for negation (unary minus) is next.	In 2 * -x, the -x is computed first, with the result then multiplied by 2.
* / %	Next to be evaluated are *, /, and %, having equal precedence.	(% is discussed elsewhere.)
+ -	Finally come + and - with equal precedence.	In y = 3 + 2 * x, the 2 * x is evaluated first, with the result then added to 3, because * has higher precedence than +. Spacing doesn't matter: y = 3+2 * x would still evaluate 2 * x first.
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right.	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

Source: Zybooks, Table 2.5.2

Top Hat Question # 8

What is the result of this expression?

(3 x 2) + (8 + 4) x 2

(a) 28
(b) 23
(c) 30
(d) 36

Expressions – integer division

- The division operator (**/**) returns a floating-point number

10 / 20	→	0.5
10 / 2		5.0

- The floored division* operator (**//**) returns:
  - a floating-point number if either operand is a float
  - an integer if both operands are integers

*used to round down the result of a floating-point division to the closes whole value

10 // 20	→	0
10 // 2		5
5.0 // 2		2.0

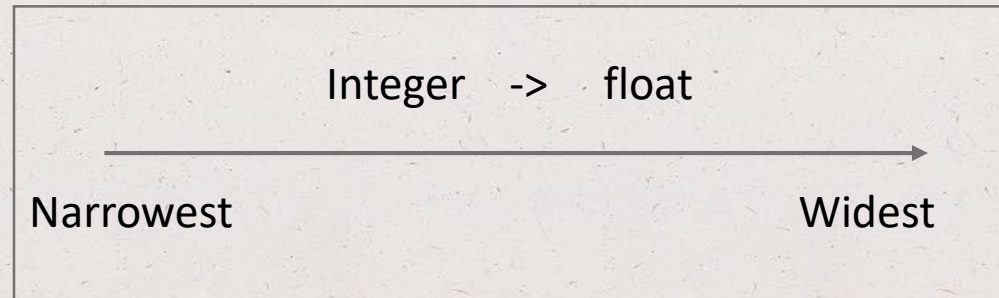# Expressions – modulo operator

- The modulo operator (**%**) evaluates the reminder of the division of two integer operands

10 % 20	→	10
10 % 2		0

Type conversion

- Expressions sometimes combine a floating-point and integer
- So what is the data type of the result of the expression in this case?

Integer -> float

Narrowest   Widest

- **Implicit conversion:** implicit cast made by the interpreter to the widest type
- **Explicit conversion:** implicit cast initiated by the programmer using data type methods, such as
  - int() converts to integer type (if the input is a float number it will chop off the decimal part)
  - float() convers to float type
  - str() converts to string type

Type conversion

- Implicit conversion:

data_type

```
print(5 + 2.0)
print(5 + 2.2)
print(5 + 2)
print (5 / 2)
print(5 / 5)
```

7.0	float
7.2	float
7	integer
2.5	float
1.0	float

- Explicit conversion:

```
print(int(5 / 5))
print(str(5 / 5))
print(float(5 + 2)
```

1	integer
1.0	string
7.0	float

Objects

- Are not created by the program!

- Instead the Python interpreter creates and manipulates objects

- Used to represent everything in Python (e.g. data types (integers, strings, lists), functions))

- Let's look at an example…

Objects

age1 = 18

Variable names

Objects in memory

98

age1 ——————→ **18**   99

Objects

age1 = 18
age2 = 23.4

Variable names

Objects in memory

Global space

Heap space

age2 → 23.4   98

age1 → 18   99

Objects

age1 = 18
age2 = 23.4
age2 = age1

Variable names

Objects in memory

Name binding:
An object can be assigned to multiple variables.
A variable cannot have multiple objects.

age2

23.4

98

age1

18

99
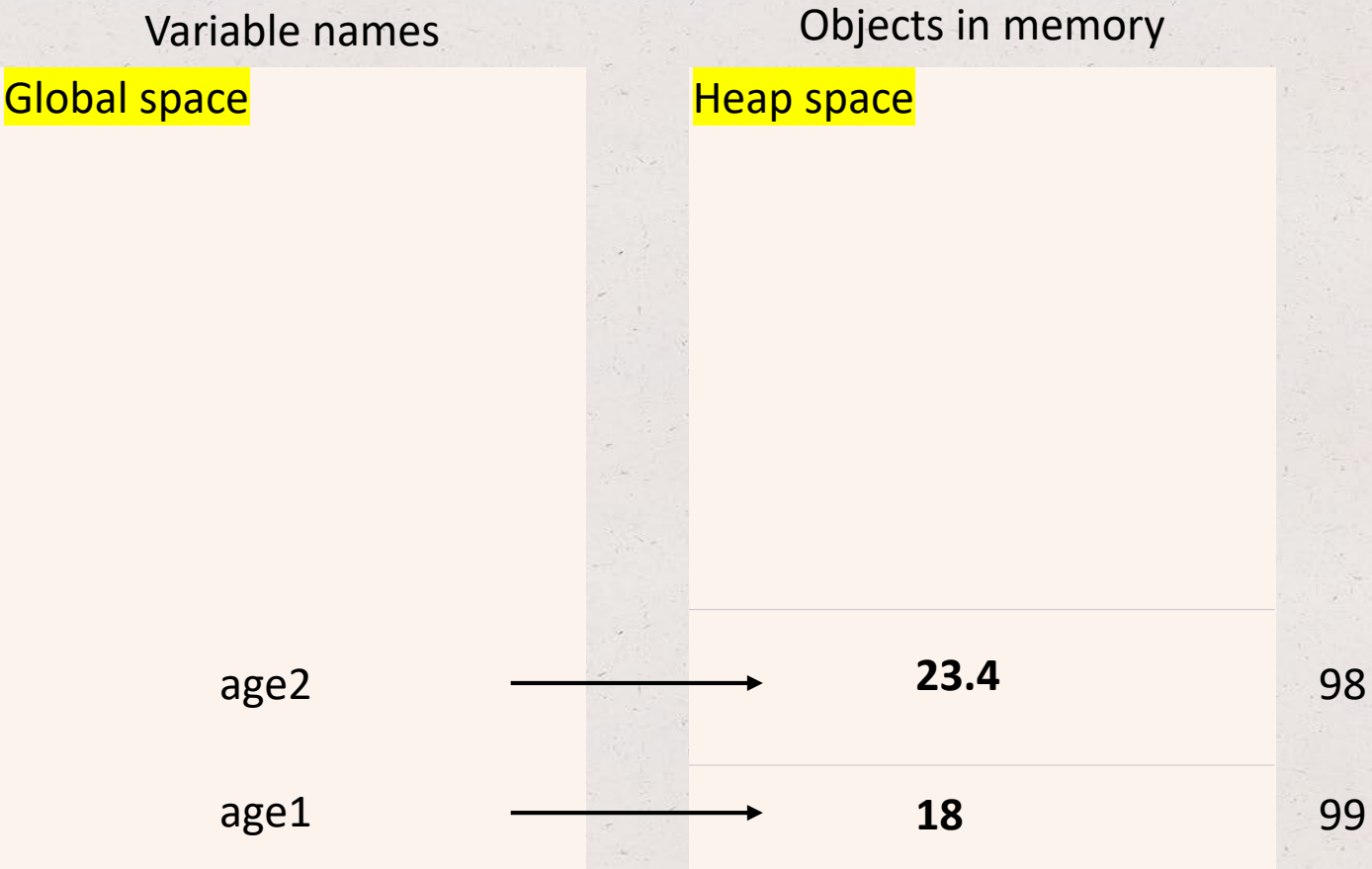
Objects

age1 = 18
age2 = 23.4
age2 = age1

Variable names

Objects in memory

Global space

Heap space

Name binding:
An object can be assigned to multiple variables.
A variable cannot have multiple objects.

age2

23.4   98

age1   **18**   99

Objects

```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
```

Variable names

Objects in memory

Global space

Heap space

	36 → 97
age2	98
age1 →	18 99

Integers and strings are **immutable** – modifying their value results in a new object being created

Objects

```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
combined_age = age1 + age2
```

Variable names

Objects in memory

Global space

Heap space

combined_age

54   96

36   97

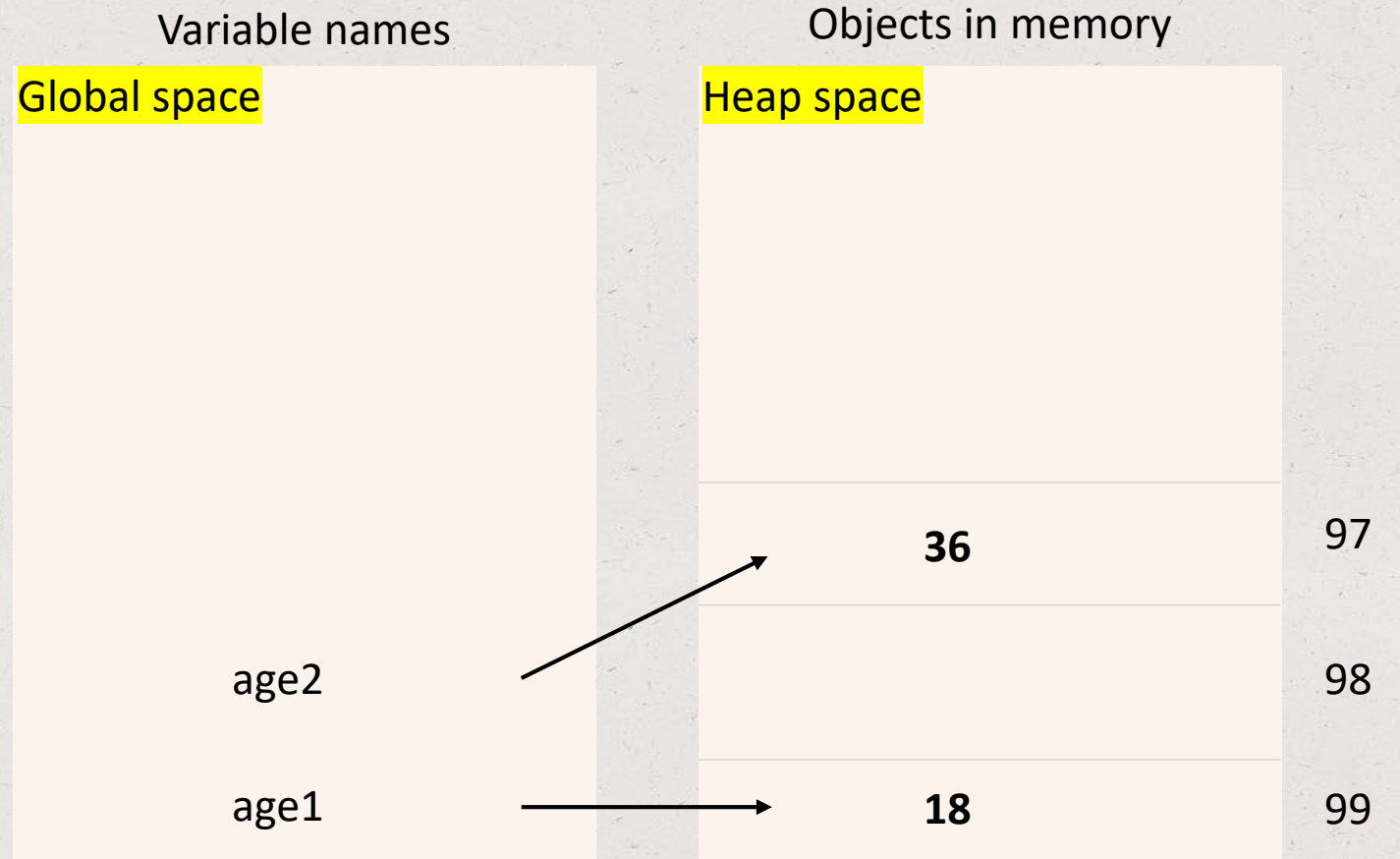age2

                      98

age1   18   99

Objects

Garbage collector (deletes objects with no reference to optimize memory)
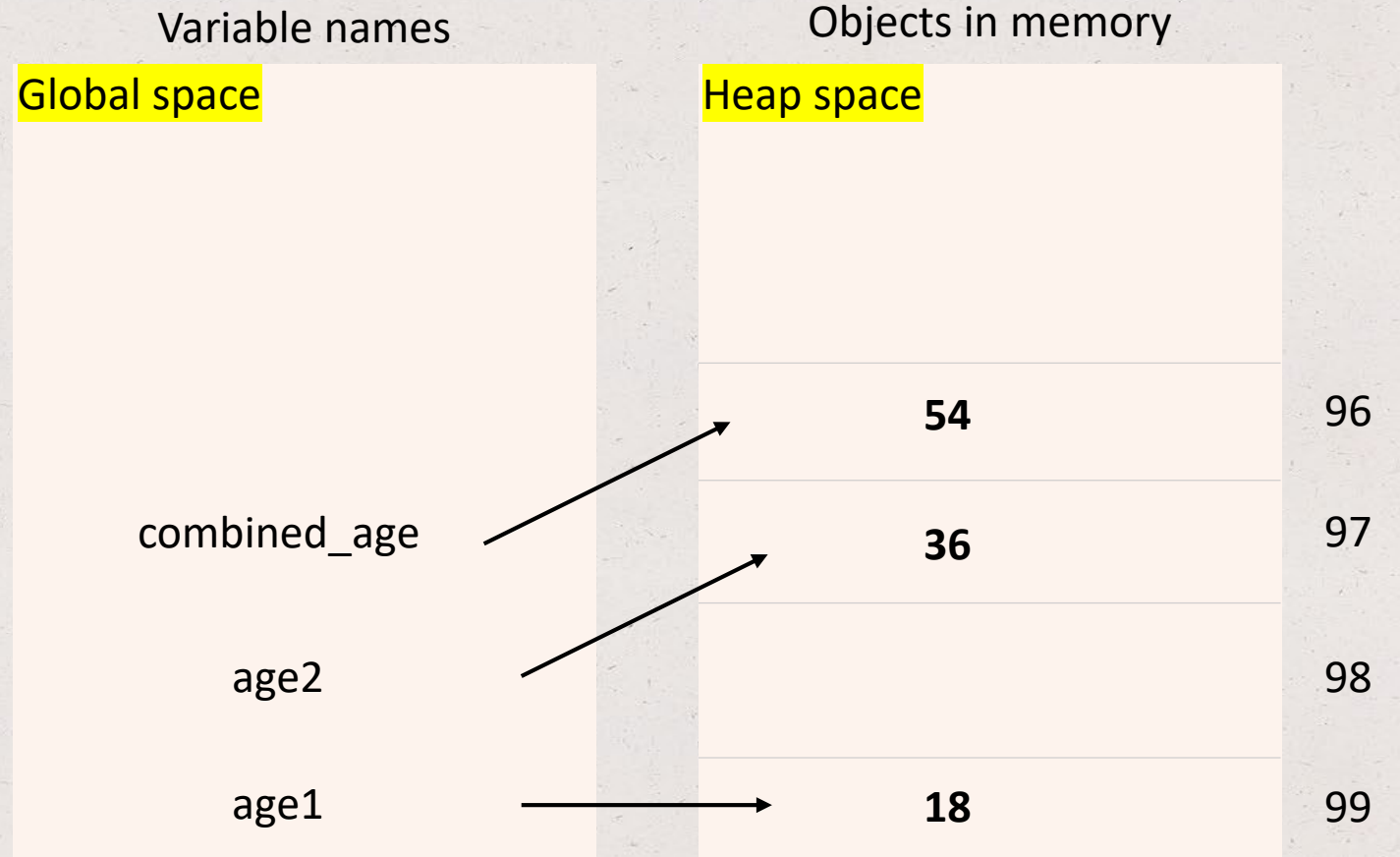
```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
combined_age = age1 + age2
```
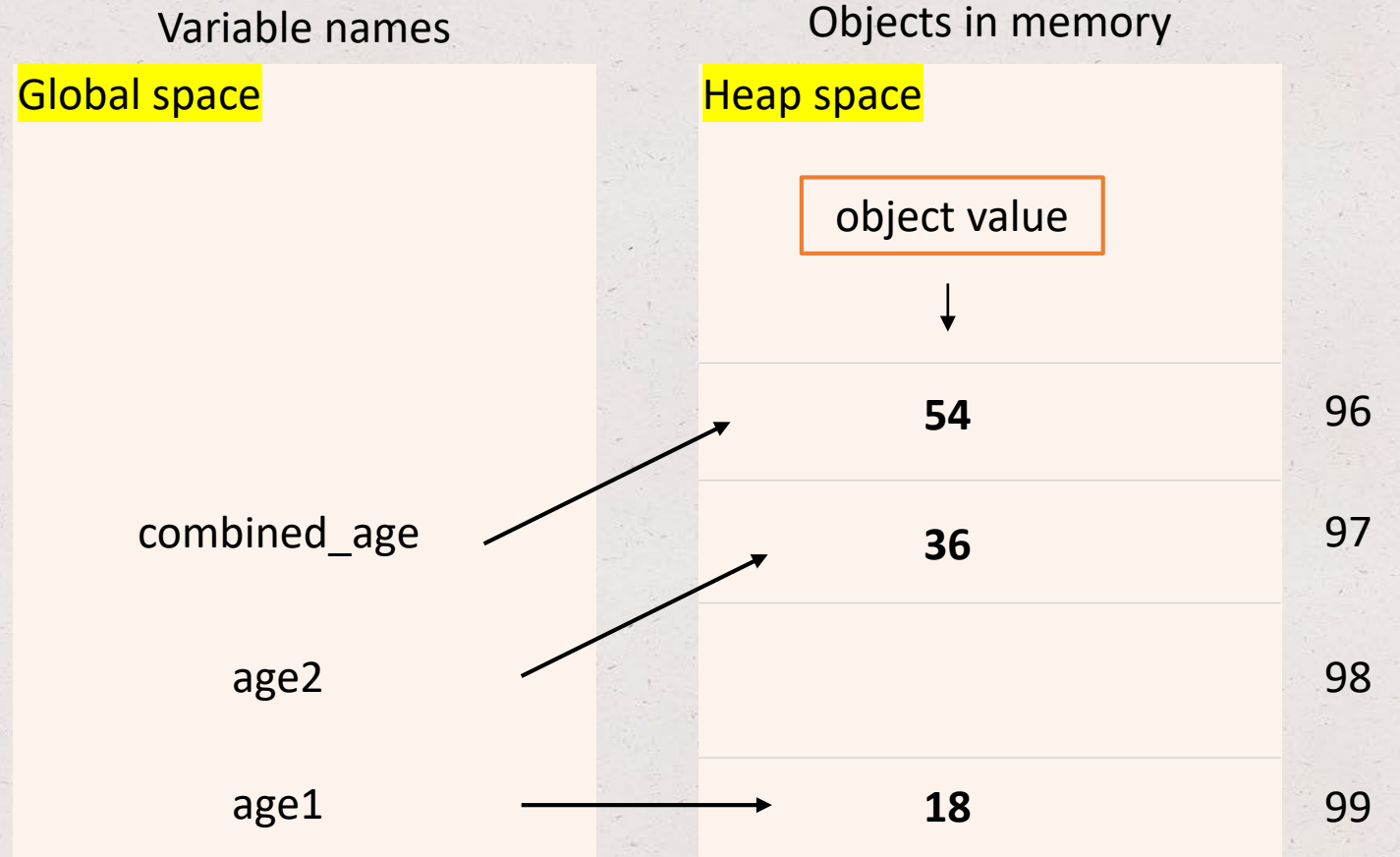
```
# to get object value
print(age1)
```
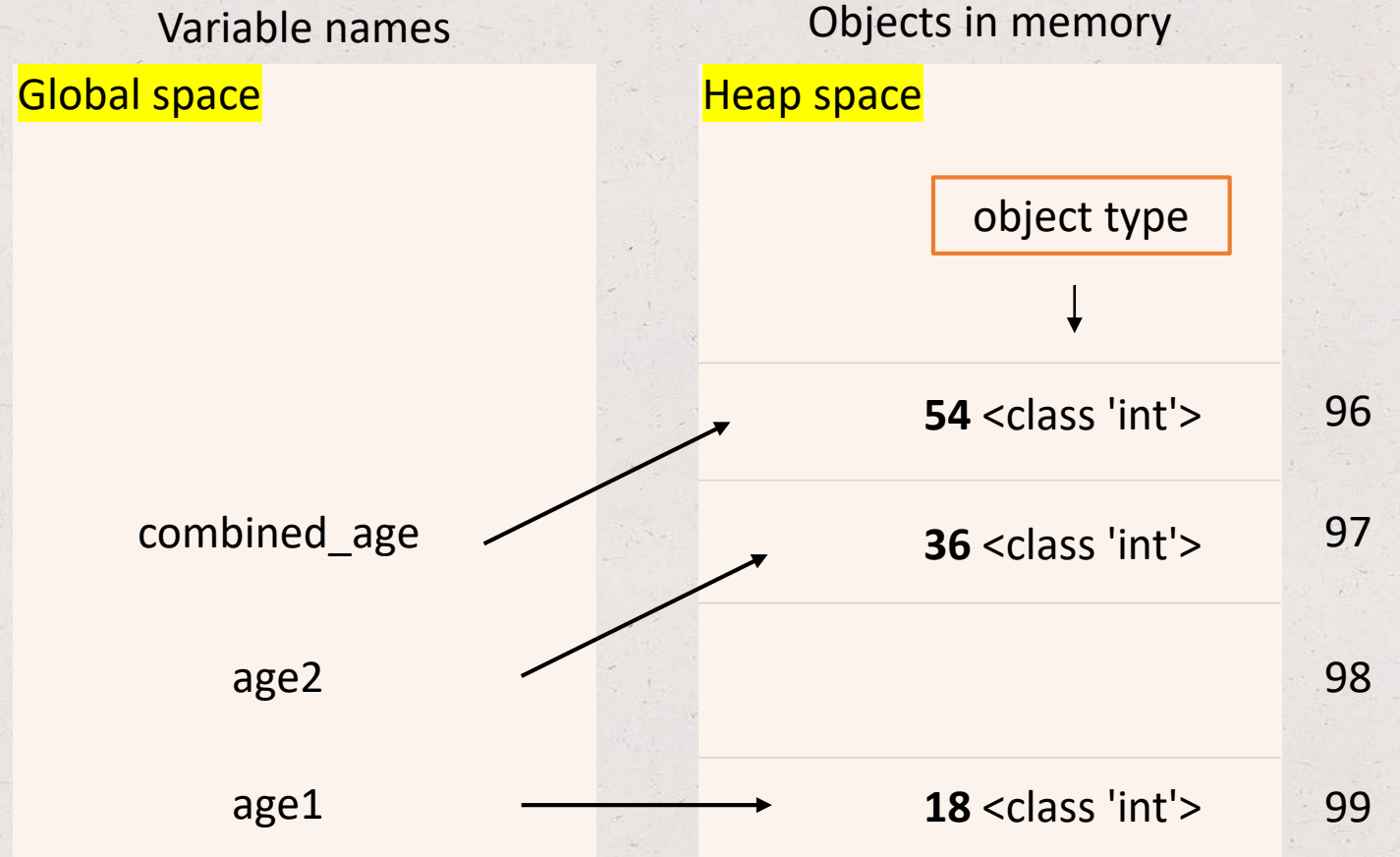
Variable names

Global space

combined_age

age2

age1

Objects in memory

Heap space

object value

54	96
36	97
	98
18	99

Objects

```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
combined_age = age1 + age2
```

```
# to get object type
print(type(age1))
```

Variable names

Objects in memory

Global space

Heap space

object type

54 <class 'int'>   96

combined_age   36 <class 'int'>   97

age2   98

age1   18 <class 'int'>   99

Arrows: age1 holds a reference to an object of type int with value 18

Objects

```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
combined_age = age1 + age2
```

```
# to get object identity (memory loc)
print(id(age1))
```

Variable names

Objects in memory

Global space

Heap space

object identity

combined_age

age2

age1

54   96

36   97

    98

18   99
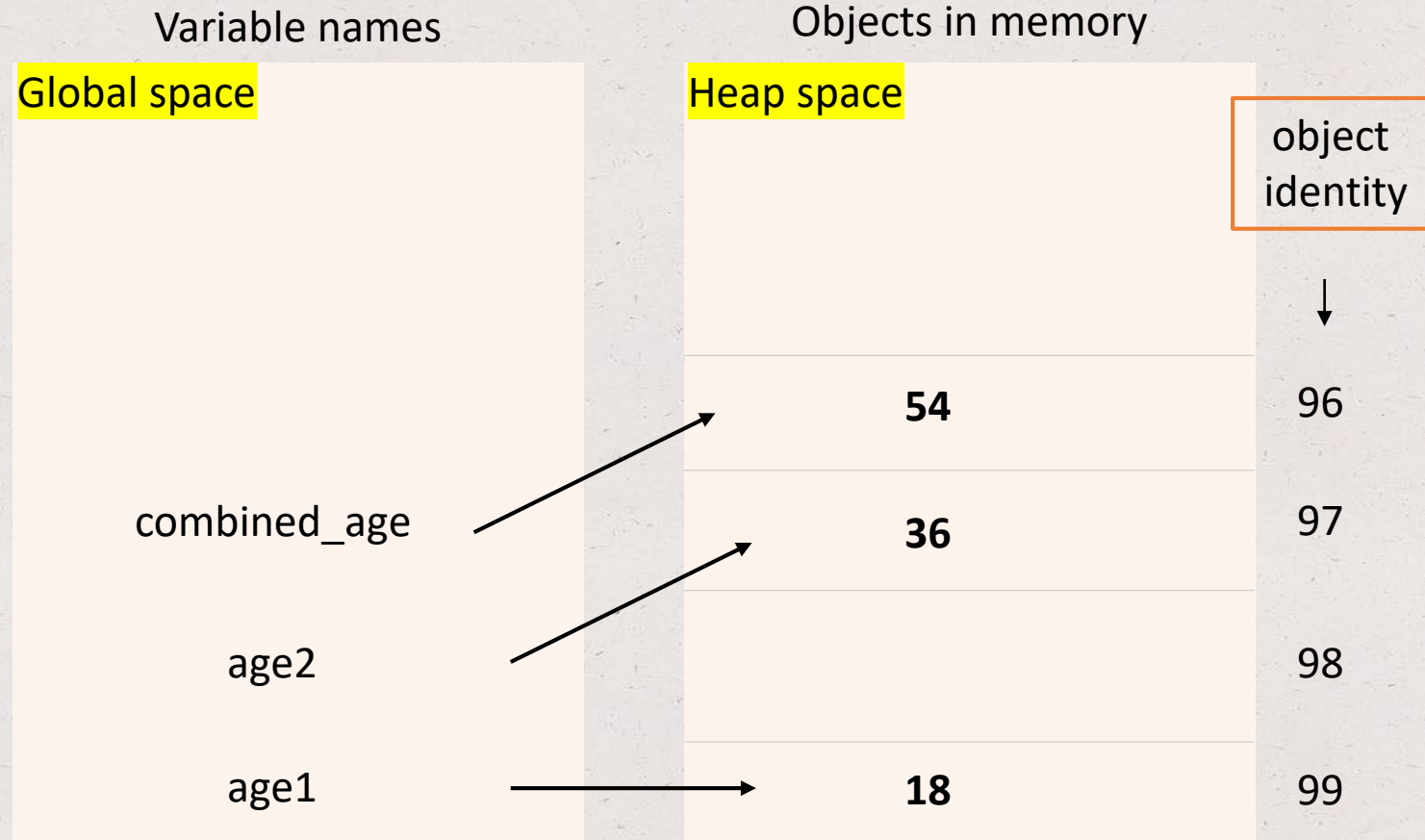
Chapter 3: Data types

- Sequence, mapping, set data types

Data types
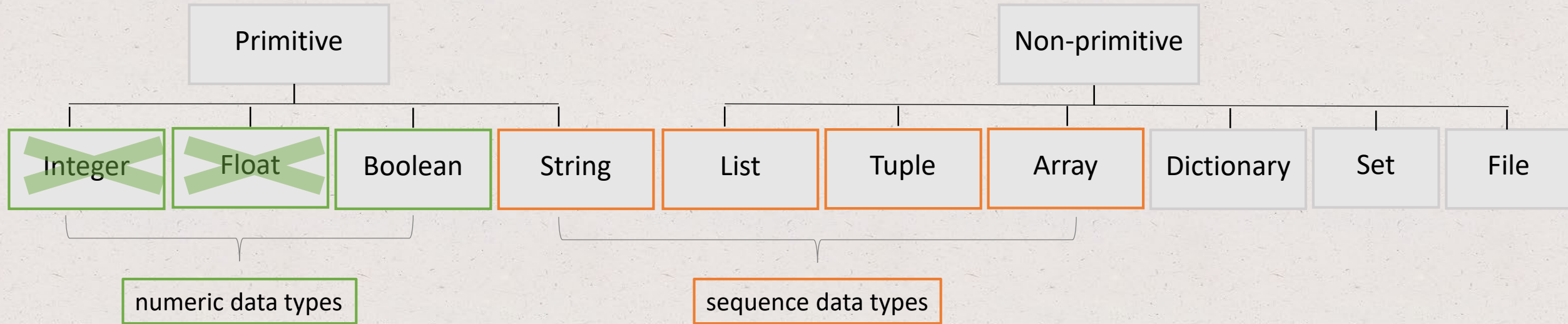
Sequence data types

- **String:** a sequence of characters; strings are immutable with fixed size

- **List**:** a container of (heterogenous) objects; lists are mutable with unlimited size

- **Tuple**:** a container of (heterogenous) objects; tuples are immutable with fixed size

** Sequence types: because objects are ordered by position (index) in the container

Note: Array is also a sequence data type but we will talk about this later in the course.

String basics

- A string is a **sequence** of characters

- A string **literal** is created by surrounding characters with single (' ') or double quotes (" ")

- The string type is a **sequence data type**, meaning that:

- The elements (characters) of a string are ordered (indexed) from left to right

0	1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		w	o	r	l	d

my_string = 'hello world'

- Indexing starts at 0

String basics – memory representation

my_string = 'Hello world'

String names

Objects in memory

Global space

Heap space

97

98

my_string   ⟶   **'Hello world'**   99

String basics

- Built-in functions:
  - len*(string)* – returns number of characters in my_string;
  - chr('a') – returns the integer 97 representing the Unicode code point of the 'a' character

- Built-in methods:
  - string.upper() – returns a copy of string with all the cased characters converted to uppercase
  - string.lower() – returns a copy of string with all the cased characters converted to lowercase
  - string.capitalize() – returns a copy of string with its first character capitalized and rest lowercased

More on Python built-in functions here:

https://docs.python.org/3.7/library/functions.html

More on Python built-in methods here:

https://docs.python.org/3/library/stdtypes.html#textseq

Top Hat Question # 9

What is the output?

```
my_string = 'Hello world'
my_number = 5
my_number = 5 * 2
print(my_string.upper())
```

String basics – accessing characters

- my_string[*index*] – character at given index (counts indexing from left to right)

- my_string[-*index*] – character at given index (counts indexing from right to left)

0	1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		w	o	r	l	d
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
my_string = 'Hello world'
print(len(my_string))
print(my_string[4])
print(my_string[-4])
```

```
11
'o'
'o'
```

String basics – add, change, remove characters

- String objects are **immutable**! Meaning that we cannot add, change, or remove a character

string1 = 'Today'
string2 = '   is 4th of July.'
string2[16] = 'n'
string2[17] = 'e'

String basics – add, change, remove characters

- String objects are **immutable**! Meaning that we cannot add, change, or remove a character
- Instead, update a character by assigning a new string

```
string1 = 'Today'
string2 = '   is 4th of July.'
string2[16] = 'n'
string2[17] = 'e'
string2 = '   is 4th of June'
```

String basics – concatenation
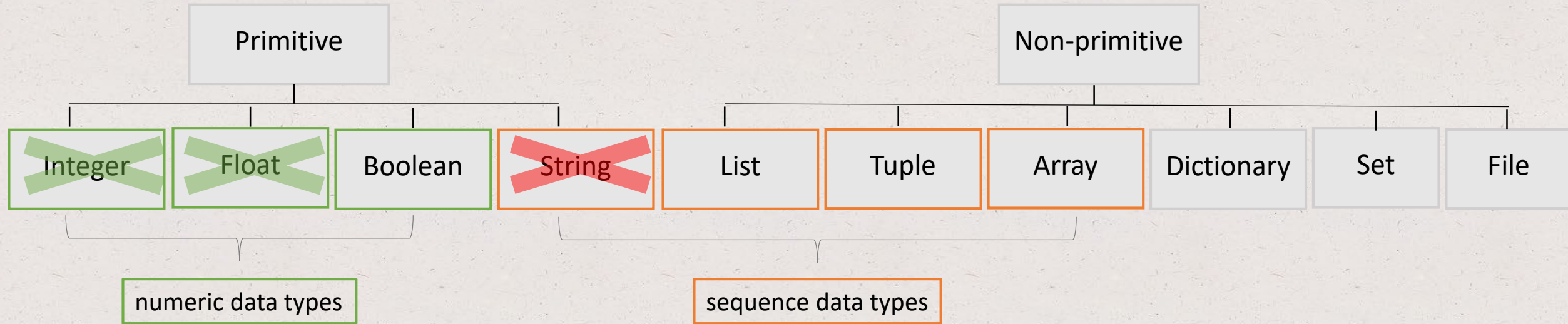
- Using the addition (**+**) sign

```
string1 = 'Today'
print(string1)
string2 = '   is 4th of July.'
print(string2)
concat_string = string1 + string2
print(concat_string)
```

```
Today

  is 4th of July

Today   is 4th of July
```

Data types

List basics

- A list is a container initialized with brackets **[ ]**
- Elements of a list are comma (**,**) separated
- Elements of a list can be heterogenous, i.e. of different types
- Elements of a list are ordered by position (index) in the container, meaning that
- A list is a **sequence data type**

- Example: my_list = ['a', 123, 'b']
- Example of empty list: my_empty_list = [ ]
- A list is **mutable**, meaning that one **can** *add, remove,* and *edit* its elements

List basics – accessing elements

- Similar to strings, list elements are accessed by index (remember list is a sequence data type)

```
my_list = ['a', 123, 'b']
print('The first element in my list is', my_list[0], '.')
```

The first element in my list is a .

List basics – add, change, remove elements

- The list data type is a **mutable object**!

- We can add, change, ore remove elements of a list using built-in methods:
    - list.append(*x*) – ads element x to the end of the list
    - list.remove(*x*) – removes the first element from the list whose value is equal to x
    - list.pop(*index*) – removes the element at the given index in the list, and returns it; if no index is specified removes and returns the last item in the list

```
my_list = ['a', 123, 'b']
my_list.append('abc')
print(my_list)
my_list.remove('a')
print(my_list)
my_list.pop(2)
print(my_list)
```

['a', 123, 'b', 'abc']

[123, 'b', 'abc']

[123, 'b']

Notice that the list output is always enclosed by brackets!

Reason: lists can be nested (more on this later)

List basics – add, change, remove elements

- The list data type is a **mutable object**!

- We can add, change, ore remove elements of a list using built-in methods:
    - list.append(x) – ads element x to the end of the list
    - list.remove(x) – removes the first element from the list whose value is equal to x
    - list.pop($index$) – removes the element at the given index in the list, and returns it; if no index is specified removes and returns the last item in the list

- Check here for more list related built-in methods:

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

List basics – concatenation

- Lists are concatenated using the (**+**) sign. Question: how do we concatenate strings?

```
list1 = ['a', 123, 'b']
list2 = ['a']
print(list2 + list1)
```

```
['a', 'a', 123, 'b']
```

Top Hat Question # 10

What is the output of the print function? Explain.

```
list1 = ['a', 123, 'b']
list2 = ['a']
list2 = list1
print(list2 + list1)
```

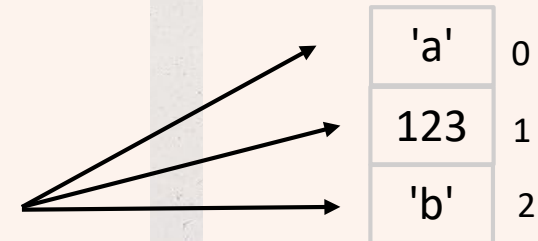# List basics – memory representation

my_list = ['a', 123, 'b']

List names

Objects in memory

The interpreter creates new object for each element in the list.
Each object is indexed

97

'a'   0

123   1

99

my_list

'b'   2

Arrow meaning: my_list holds references to objects in list

List basics

- Built-in functions:
  - len(*list*) – returns the length of the list
  - min(*list*) – returns the element with the smallest value in the list (numeric types only)
  - max(*list*) – returns the element with the largest value in the list (numeric types only)
  - sum(*list*) – returns the sum of all elements of a list (numeric types only)

- More built-in methods:
  - list.index(*x*) – returns the index of the first element in the list whose value is equal to x
  - list.count(*x*) – returns the number of times x appears in the list

More on Python built-in functions here:

https://docs.python.org/3.7/library/functions.html

More on Python built-in methods here:

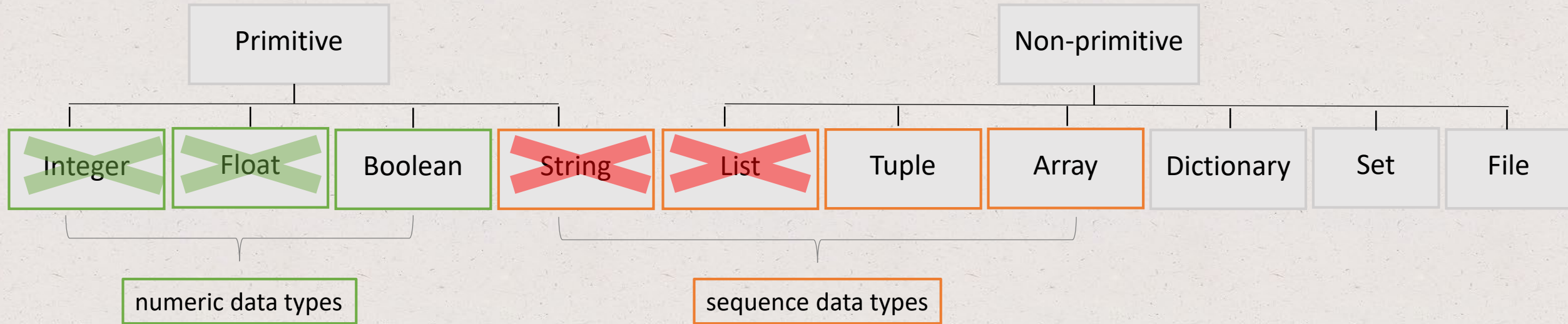https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Top Hat Question # 11

What is the output?

```
list1 = ['a', 123, 'b']
list1.count('a')
```

Data types

Tuple basics

- A tuple is a container initialized with parenthesis () or no parenthesis (safe to add them)
- Elements of a tuple are comma (**,**) separated
- Elements of a tuple can be heterogenous, i.e. of different types
- Elements of a tuple are ordered by position (index) in the container, meaning that:
- A tuple is a **sequence data type**
- Looks pretty much like a list but is more memory efficient

- Example: my_tuple = ('a', 123, 'b')
- Example of empty list: my_empty_tuple = ()
- A tuple is **immutable**, meaning that one **cannot** *add, remove,* and *edit* its elements

Tuple basics – accessing elements

- Similar to lists (and strings), tuple elements can be **accessed by index**

```
my_tuple = ('a', 123, 'b')
print('The first element in my list is', my_tuple[0], '.')
print(my_tuple)
```

```
The first element in my list is a .
('a', 123, 'b')
```

Notice that the tuple output is always enclosed by brackets!

Reason: tuples can be nested (more on this later)

Tuple basics – accessing elements

- Similar to lists (and strings), tuple elements can be **accessed by index**

my_tuple = ('a', 123, 'b') print('The first element in my list is', my_tuple[0], '.') print(my_tuple)	The first element in my list is a . ('a', 123, 'b')

- **When to use tuples?** When you want to make sure that the values do not change. Remember that tuples are immutable objects, so you cannot change, add, or delete values.

Tuple basics – memory representation
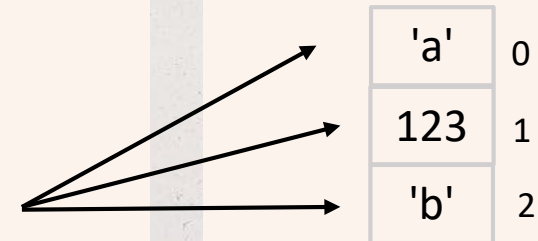
my_tuple = ('a', 123, 'b')

Tuple names

Objects in memory

The interpreter creates new object for each element in the tuple. Each object is indexed

97

my_tuple

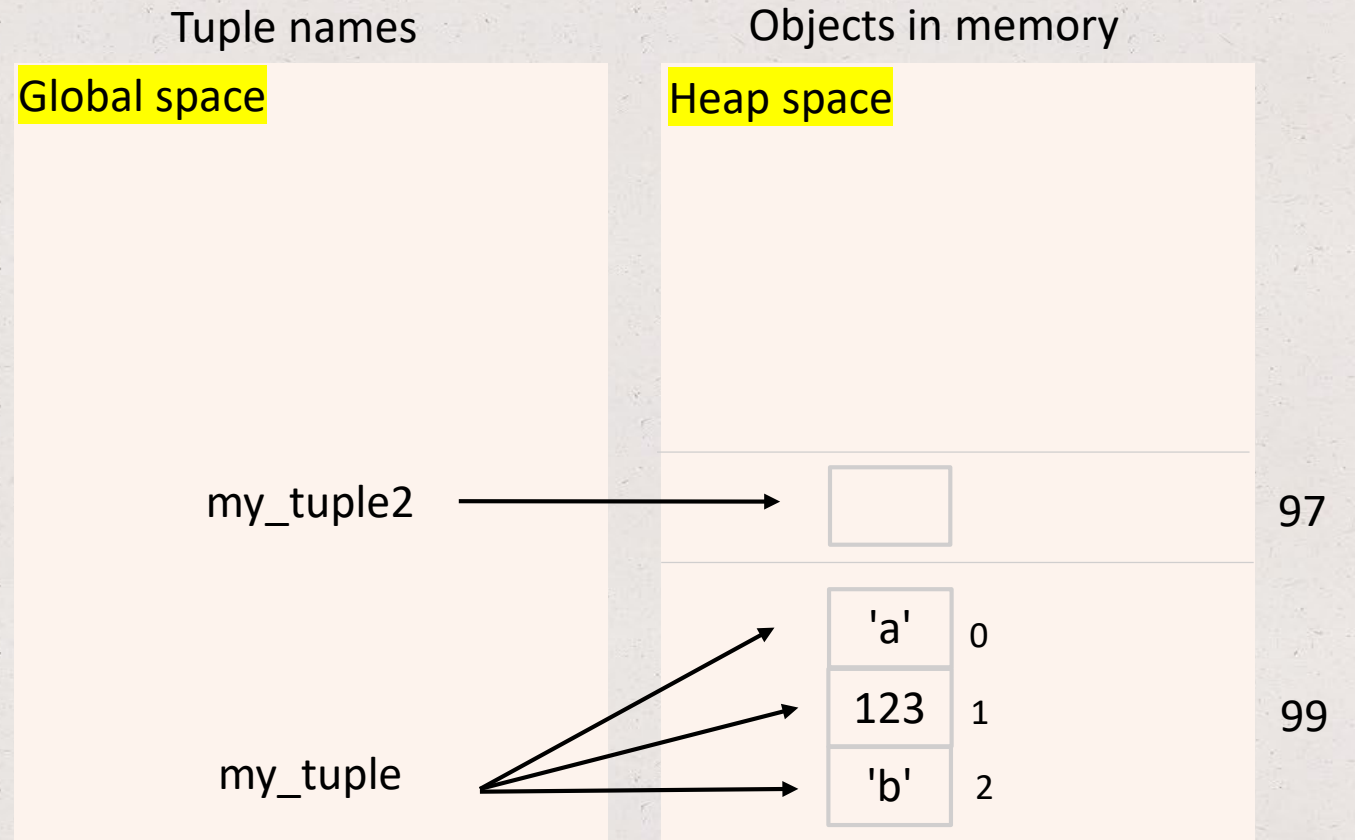| 'a' | 0 |
| 123 | 1 |
| 'b' | 2 |

99

Arrow meaning: my_tuple holds references to objects in tuple

Tuple basics – memory representation

my_tuple = ('a', 123, 'b')
my_tuple2 = ()

Tuple names

Objects in memory

my_tuple2 →   97

                          'a'   0

                          123   1   99
my_tuple
                          'b'   2

Arrow meaning: my_tuple holds references to objects in tuple.
my_tuple2 is an empty set

Data types
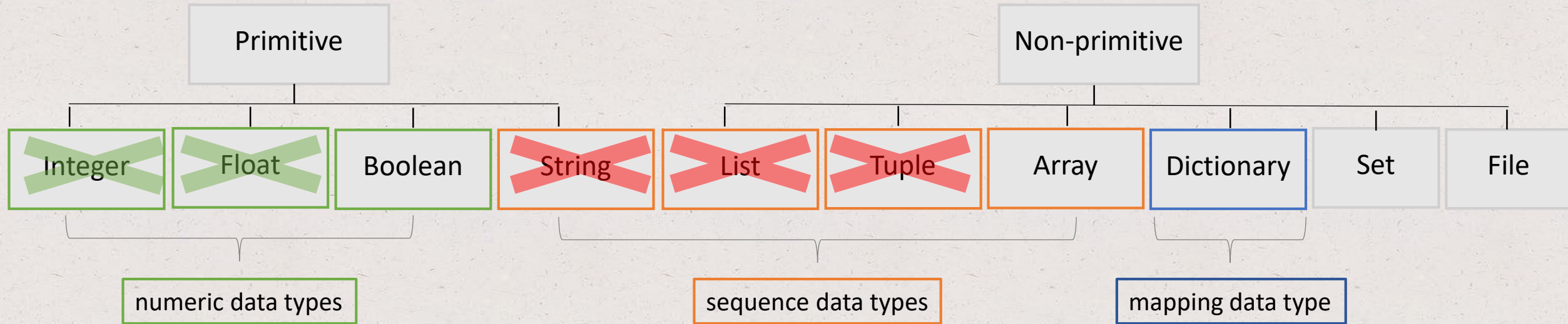
Dictionary basics

- A dictionary is a container used to describe associative relationships between **keys** (words) and **values** (definitions)

- It's initialized with curly brackets **{}** that surround the **key : value** pairs

- Multiple **key : value** pairs are separated by comma (**,**)

- The keys can be any *immutable type*, e.g. numeric, string, tuple

- The values can be of *any type*

- Example: airplanes= {'boeing737' : 134.9, 'boeing777' : 442.2}

dictionary name

Dictionary basics

- A dictionary is a container used to describe associative relationships between **keys** (words) and **values** (definitions)

- It's initialized with curly brackets **{}** that surround the **key : value** pairs

- Multiple **key : value** pairs are separated by comma (**,**)

- The keys can be any *immutable type*, e.g. numeric, string, tuple

- The values can be of *any type*

- Example: airplanes= {'boeing737' : 134.9, 'boeing777' : 442.2}

        ↑   ↑

       key   key

Dictionary basics

- A dictionary is a container used to describe associative relationships between **keys** (words) and **values** (definitions)

- It's initialized with curly brackets **{}** that surround the **key : value** pairs

- Multiple **key : value** pairs are separated by comma (**,**)

- The keys can be any *immutable type*, e.g. numeric, string, tuple

- The values can be of *any type*

- Example: airplanes= {'boeing737' : 134.9, 'boeing777' : 442.2}

                          ↑   ↑

                        value   value

Dictionary basics

- A dictionary is a container used to describe associative relationships between **keys** (words) and **values** (definitions)

- It's initialized with curly brackets **{}** that surround the **key : value** pairs

- Multiple **key : value** pairs are separated by comma (**,**)

- The keys can be any *immutable type*, e.g. numeric, string, tuple

- The values can be of *any type*

- Example: airplanes= {'boeing737' : 134.9, 'boeing777' : 442.2}

- Example empty dictionary: airplanes = { }

Dictionary basics

- A dictionary is a **mapping data type**, so not a sequence data type (!)
- This means that elements of a dictionary do not maintain any specific ordering
- A dictionary is a **mutable** object meaning that one can *add, remove,* and *edit* its elements

products = {'GM seed': 120, 'Conv' : 87}
print(products)

{'GM seed' : 120, 'Conv : 87'}

Notice that the dictionary output is always enclosed by curly brackets!

Reason: dictionaries can be nested (more on this later)

Dictionary basics – accessing elements

- Dictionary elements can only be **accessed by key name** (no index -> no ordering of entries)

products = {'GM seed': 120, 'Conv' : 87} print('The price of GM seed is', products['GM seed'])	The price of GM seed is 120

- What if you try to access a key that doesn't exist in the dictionary?

products = {'GM seed': 120, 'Conv' : 87} print('The price of GM seed is', products['IR seed'])	Traceback (most recent call last): File "<stdin>", line 2, in <module> KeyError: 'IR seed'

Dictionary basics - add, change, remove elements

- The dictionary data type is a **mutable object**!

- To **add** a **key : value** pair in the dictionary, assuming the key : value pair doesn't already exist

```
products['IR seed'] = 110
```

- To **change** a **value** *belonging* to an **existing key**

```
products['Conv'] = 50
```

- To **remove** a **key**, assuming the key already exists

```
del products['GM seed']
```

Dictionary basics – memory representation

```
airplanes = {
    'boeing737' : 134.9,
    'boeing777' : 442.2
}
```

Dictionary name

Global space

Objects in memory

Heap space

98

airplanes

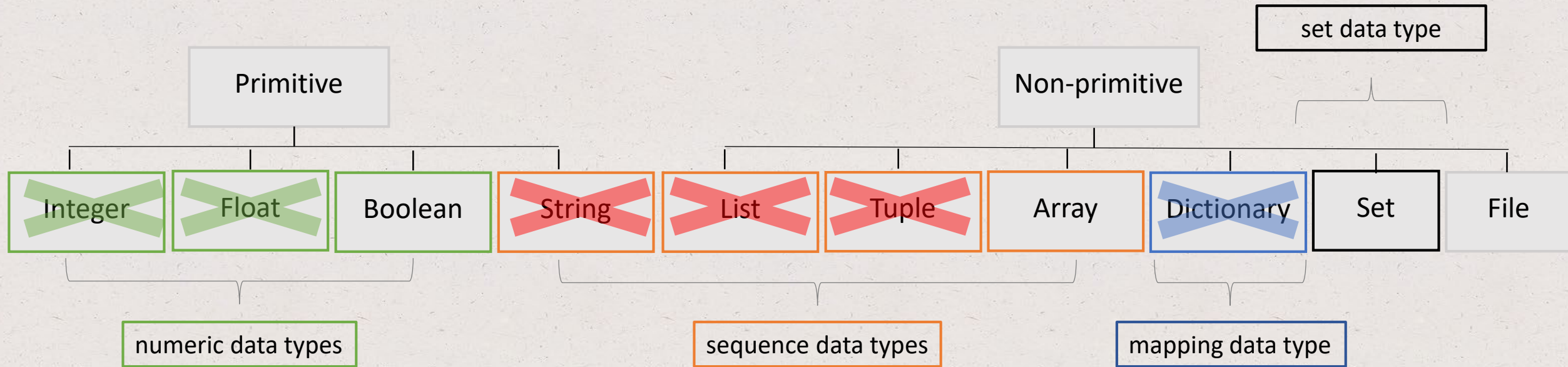| 134.9	'boeing737'
442.2	'boeing777'

99

Top Hat Question # 12

Write code to compute the sum of the price of boeing 737 and boeing 777

```
airplanes = {
    'boeing737' : 134.9,
    'boeing777' : 442.2
}
```

Data types

Set basics

- A set is a container of **unordered** (no indexing!) and **unique** elements
- Initialization can be done in two ways:
  - Using the built-in set() function, which accepts only sequence-type objects (string, list, tuple)
  - Using the set literal with curly brackets **{}**; elements are separated by a comma (**,**)

- **Unique** elements: duplicate values are removed when passed into the set
- Sets are **mutable objects**, meaning that one can add or remove elements

- Example using the set() function: my_set = set(['a', 'b', 'c'])
- Example using the set literal: my_set = {'a', 'b', 'c'}
- Example of an empty set: my_set = set( ) -> note: an empty set can be only created with set()
- If you type my_set = { }, then an empty dictionary will be created!

Set basics

- More examples

```
my_set = set(['a', 'b', 'c', 'd', 'c', 'a'])
print(my_set)
```

```
{'c', 'a', 'b'}
```

No specific ordering. Run the code again and you might see a different ordering

Notice that the set output is always enclosed by curly brackets!

Reason: sets can be nested (more on this later)

Set basics – adding or removing elements

- The set data type is a **mutable object**!

- We can add or remove elements of a set using methods included in the set() function:
    - set.add(*x*) – add element x to *set*
    - set.remove(*x*) - remove element x from *set*; raises KeyError if not present
    - set.pop() – remove and return an arbitrary element from *set*; raises KeyError if set is empty
    - set.clear() – remove all elements from *set*; the resulting set is an empty set (of length 0)
    - set.update(*set_a*) – returns *set* with elements added from *set_a*

- Check here for more set related objects here:

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Set basics – memory representation

airplanes = set(['boeing 737', 'boeing 777'])

Set names

Global space

Objects in memory

Heap space

98

airplanes ⟶ 'boeing 737'   'boeing 777'   99

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
```

Set names

<mark>Global space</mark>

Objects in memory

<mark>Heap space</mark>

cars ⟶ | 'VW Tiguan' | 98

airplanes ⟶ | 'boeing 737'   'boeing 777' | 99

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
```

Variable names

Objects in memory
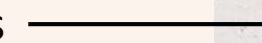
Global space

Heap space

cars ⟶ | 'VW Tiguan'   'VW Passat' | 98

airplanes ⟶ | 'boeing 737'   'boeing 777' | 99

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
```

Variable names

Objects in memory

Global space

Heap space

cars →

'VW Tiguan'   'VW Passat'   98
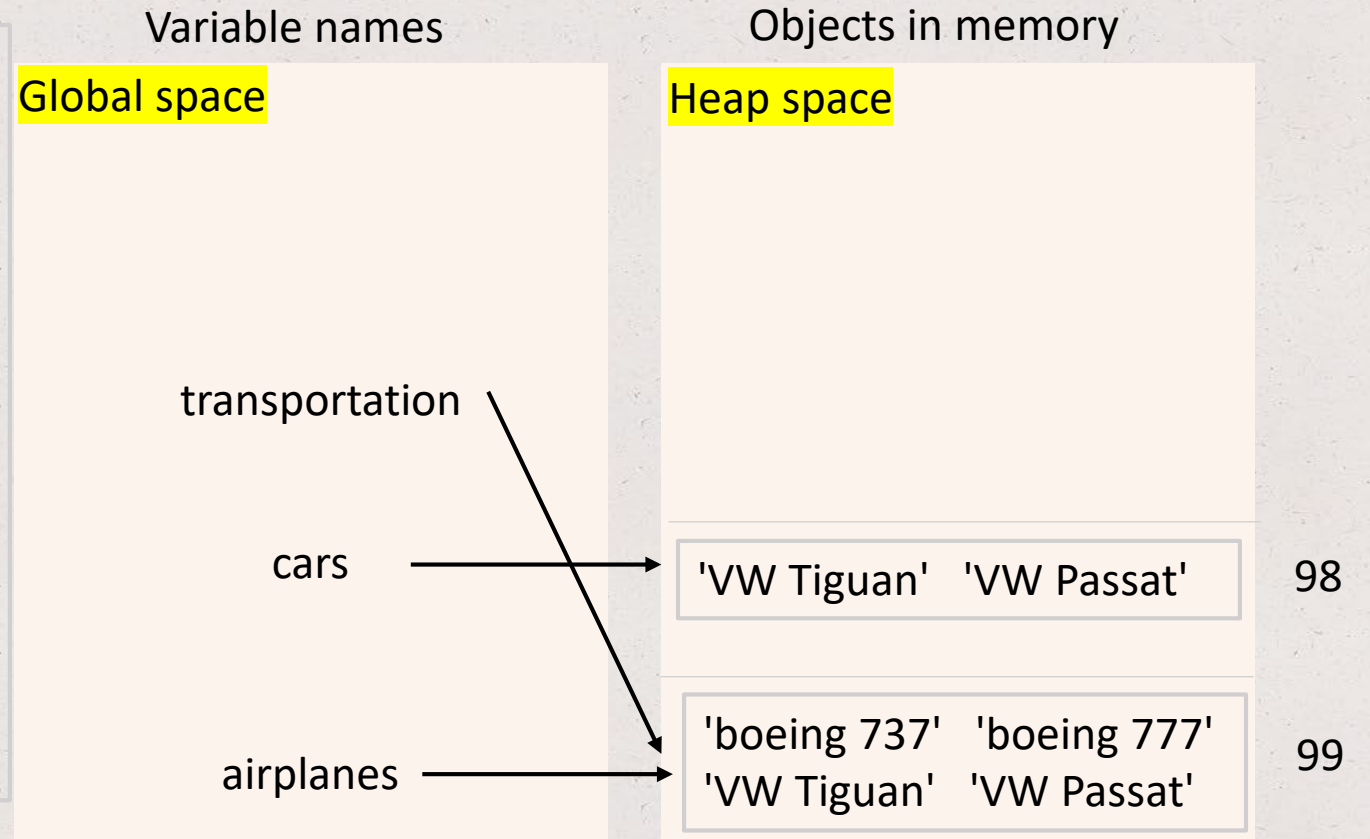
airplanes →

'boeing 737'   'boeing 777'
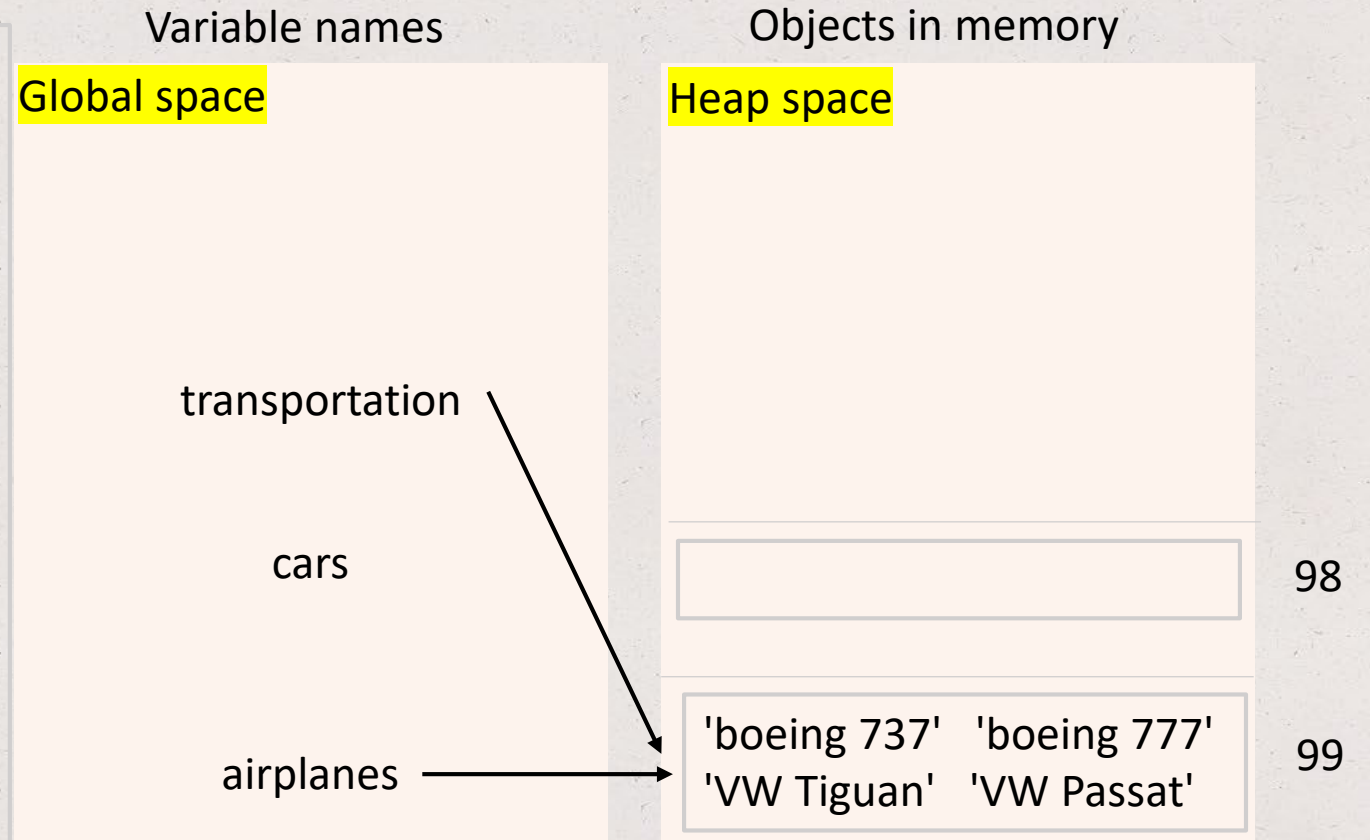'VW Tiguan'   'VW Passat'   99

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
transportation = airplanes
```

Variable names

Objects in memory

Global space

Heap space

transportation

cars → 'VW Tiguan'   'VW Passat'   98

airplanes → 'boeing 737'   'boeing 777'
'VW Tiguan'   'VW Passat'   99

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
transportation = airplanes
cars.clear()
```

Variable names

Objects in memory

Global space

Heap space

transportation

cars

98

airplanes

'boeing 737' 'boeing 777'
'VW Tiguan'   'VW Passat'

99

Note: cars is now an empty set

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
transportation = airplanes
cars.clear()
airplanes.clear()
```

Variable names

<mark>Global space</mark>

transportation

cars

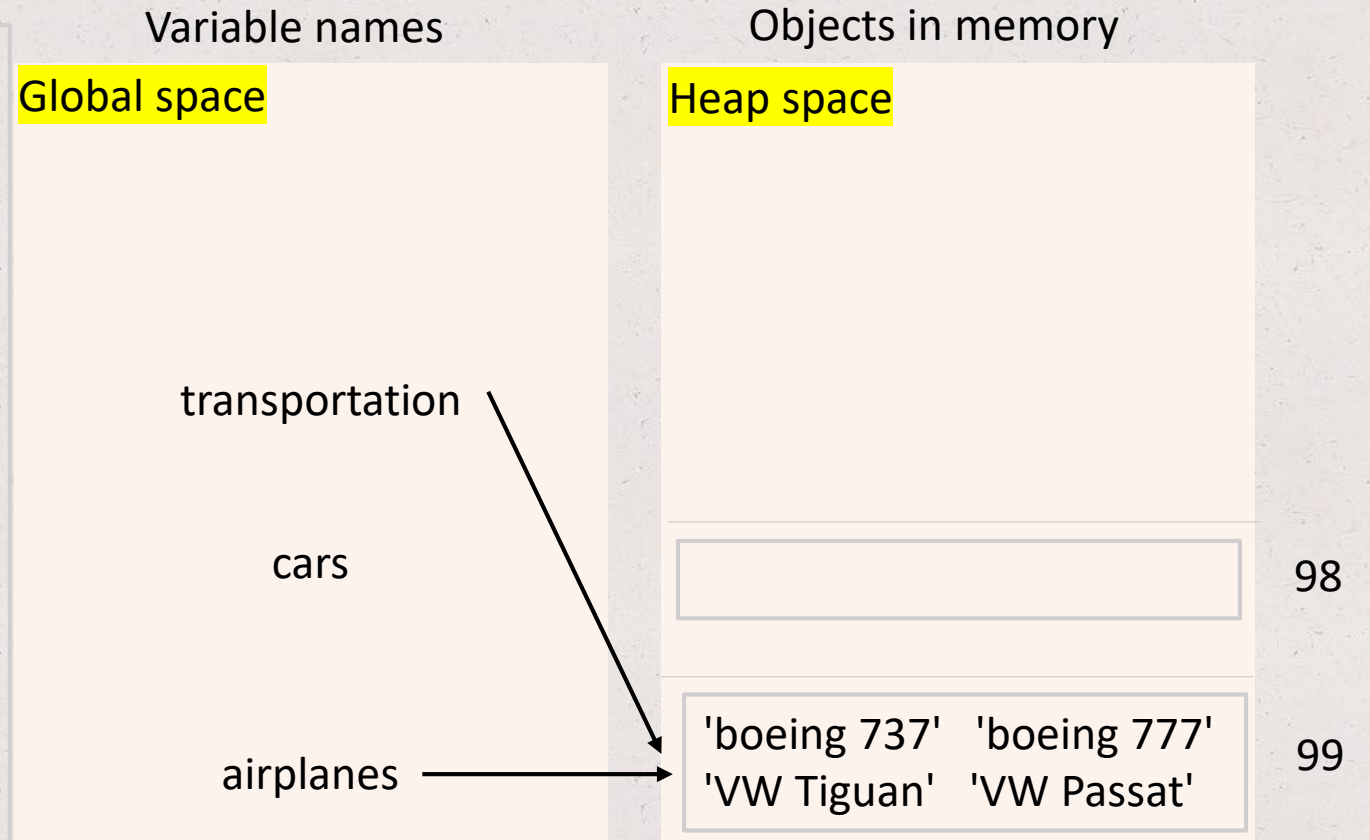airplanes

Objects in memory

<mark>Heap space</mark>

98

99

Note: cars, transportation, airplanes are now empty sets

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
transportation = airplanes
cars.clear()
```

Variable names

Global space

transportation

cars

airplanes

Objects in memory

Heap space

98

'boeing 737'   'boeing 777'
'VW Tiguan'   'VW Passat'

99

Note: cars is now an empty set

Set basics – memory representation

```
airplanes = set(['boeing 737', 'boeing 777'])
cars = set(['VW Tiguan'])
cars.add('VW Passat')
airplanes.update(cars)
transportation = airplanes
cars.clear()
transportation.clear()
```

Variable names	Objects in memory
Global space	Heap space

transportation

cars

airplanes

98

99

Note: cars, transportation, airplanes are now empty sets
Empty sets have **len(set) = 0**

Set basics – set theory operations

- The set data type supports common set theory operations, such as:

- We can add or remove elements of a set using methods included in the set() function:
  - set.intersection(*set_a, set_b*) – returns a set with all elements in common between *set* and *set_a, set_b*
  - set.union(*set_a, set_b*) – returns a set containing all of the unique elements in all sets

- Check here for more set related built-in objects here:

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
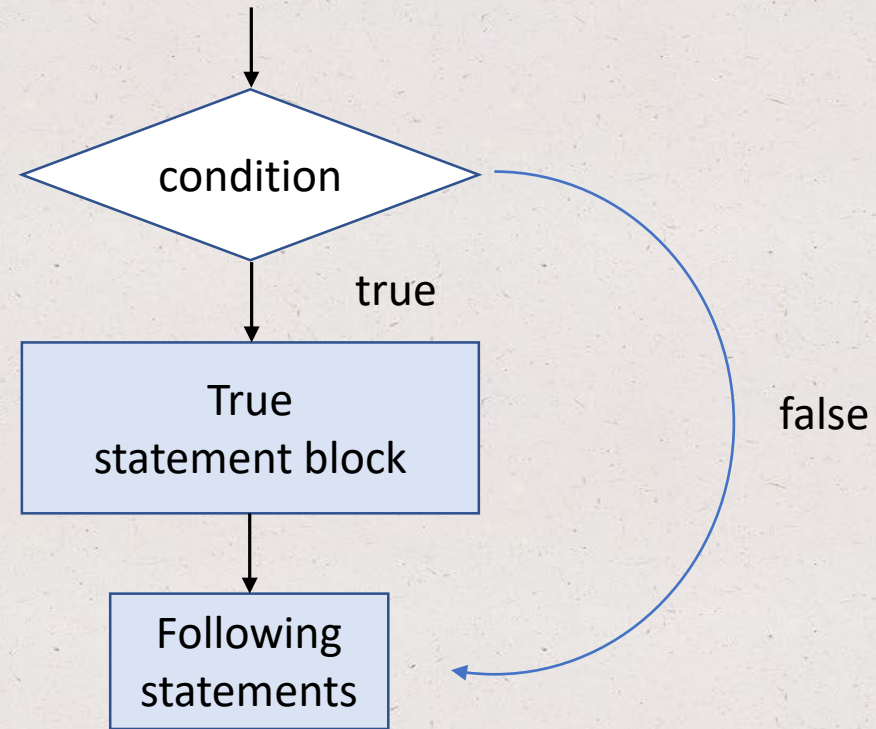
Chapter 4: Branches

- Conditional statements

- Boolean statements

- Code blocks and indentation

- Conditional expressions

Conditional statements
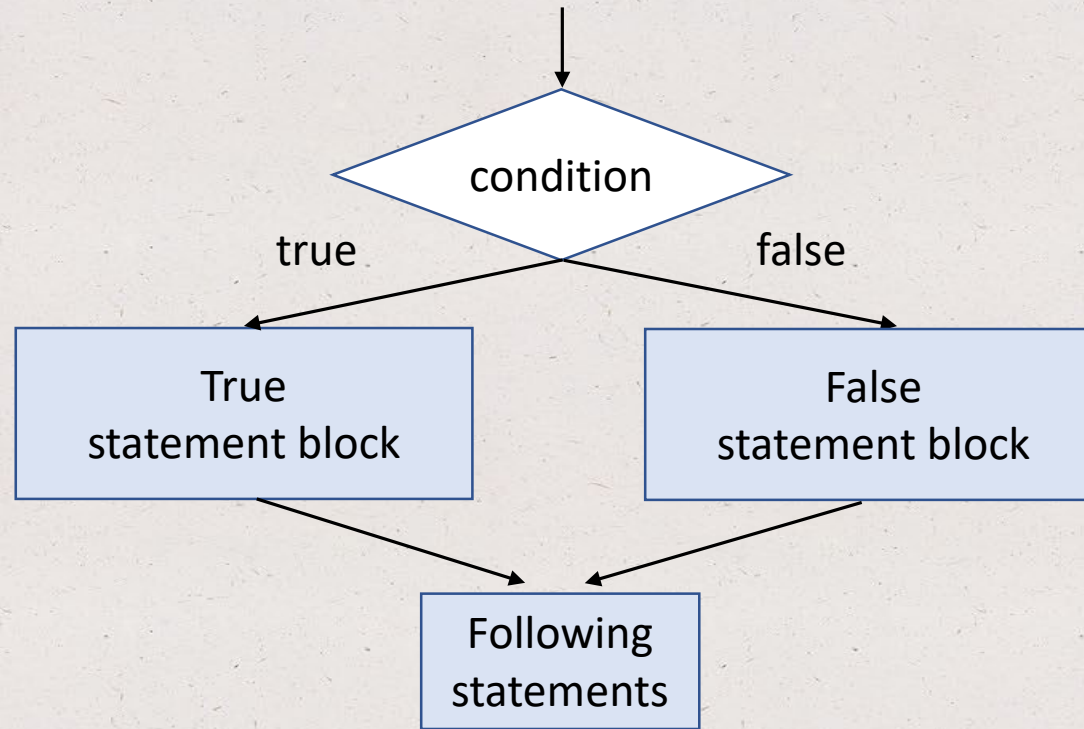
- The **if** statement:

```
if condition:
    true statement
    .
    .
    last true statement
```

Conditional statements

- The **if-else** statement:
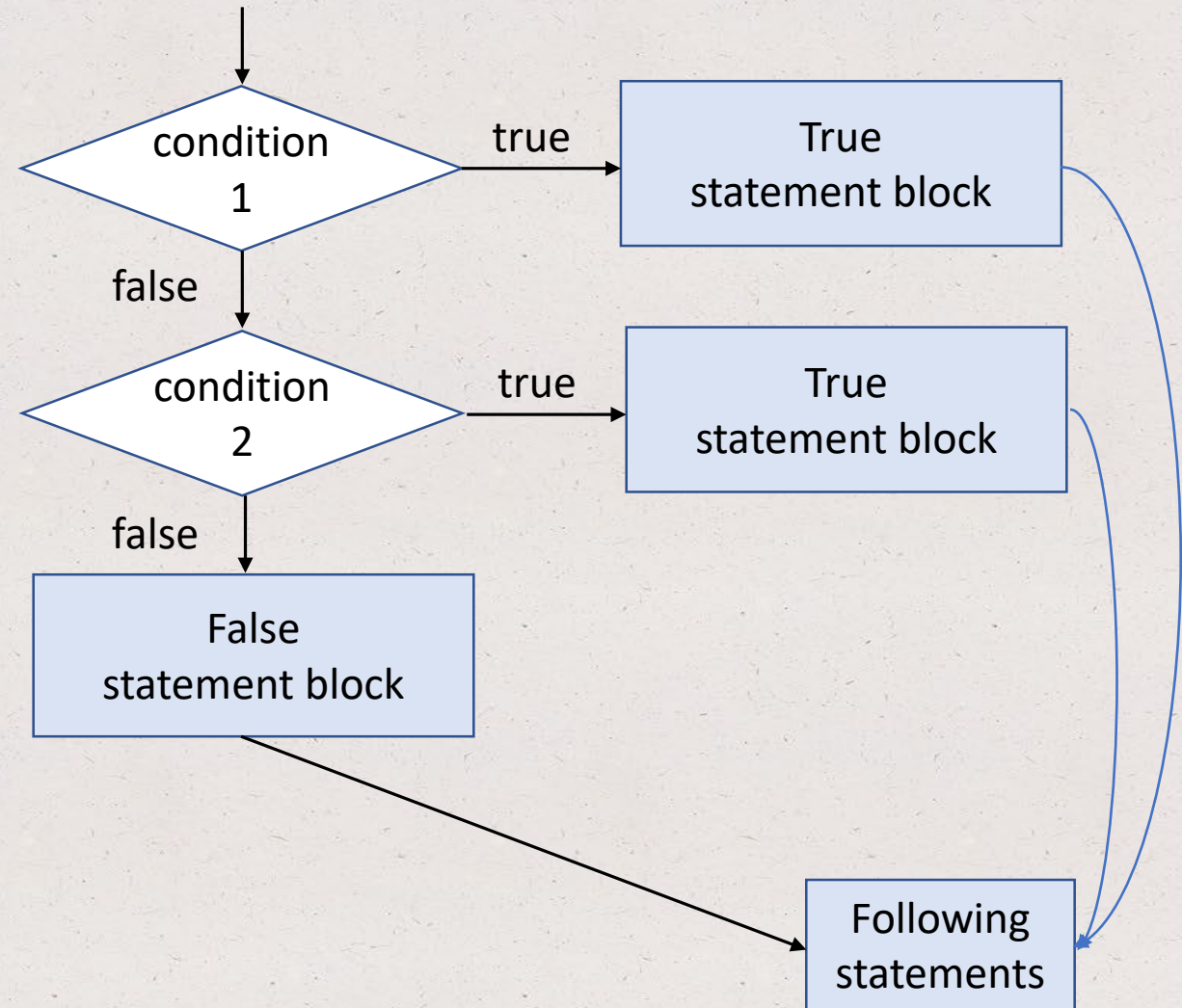
```
if condition:
    true statement
    .
    .
    last true statement
else:
    false statement
    .
    .
    false statement
```

Conditional statements

- The **if-else if** statement:

```
if condition1:
    true statement

    .

    .

    last true statement
elif condition2:
    true statement

    .

    .

    true statement
else:
    false statement

    .

    .

    false statement
```
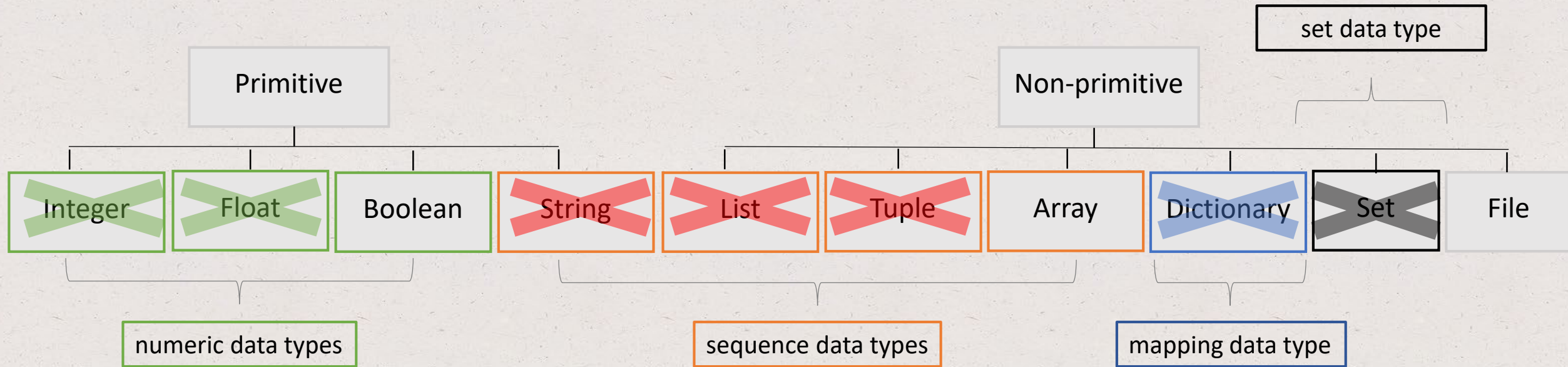
Boolean statements

- Boolean values: *True* or *False* (capitalized!)
- Boolean operators: *and, or, not*
  - **and**: true when both operands are true
  - **or**: true when at least one operand is true
  - **not**: True (False) when the single operand is False (True)
- Boolean expression: an expression that uses Boolean operators
- Examples of Boolean expressions:

| age = 18, days = 7 | (age > 16) **and** (age <25)
(age < 16) **and** (days < 8)
(age > 16) **or** (days > 7)
not (days > 6) | True **and** True -> True
False **and** True -> False
True **or** False -> True
not (True) -> False |

Data types

Boolean statements

- **Comparison operators:** (expression1 **comparison operator** expression2), e.g. (age > 16)
  - *Equality comparison:*
    - ==: equal to
    - !=: not equal to
  - *Relational comparison:*
    - >: greater than
    - <: less than
    - >=: greater than or equal
    - <=: less than or equal
    - a < b < c: operator chaining (b > a?, b < c?; note that a is never compared to c)

Boolean statements

- Why are Boolean operators (**and**, **or**, **not**) important in programming?

Boolean statements

- Why are Boolean operators (**and**, **or**, **not**) important in programming?

- They are commonly used in expressions found in if-else statements!

- To understand loops you need to understand Booleans.

- How do you code "*I cannot rent a car if I am less than 21 years old and more than 81 years old*" using Boolean operators?

```
age = int(input('Enter driver\'s age: ')
if (age >= 21) and (age <= 81):
    print('I can rent a car')
else:
    if (age < 21):
        print('Too young to rent a car')
    else:
        print('Too old to rent a car')
```

Boolean statements – operator precedence

- An expression is evaluated using the order of standard mathematics

Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first	In `(a * (b + c)) - d`, the + is evaluated first, then *, then -.
* / % + -	Arithmetic operators (using their precedence rules; see earlier section)	`z - 45 * y < 53` evaluates * first, then -, then <.
< <= > >= == !=	Relational, (in)equality, and membership operators	`x < 2 or x >= 10` is evaluated as `(x < 2) or (x >= 10)` because < and >= have precedence over or.
not	not (logical NOT)	`not x or y` is evaluated as `(not x) or y`
and	Logical AND	`x == 5 or y == 10 and z != 10` is evaluated as `(x == 5) or ((y == 10) and (z != 10))` because and has precedence over or.
or	Logical OR	`x == 7 or x < 2` is evaluated as `(x == 7) or (x < 2)` because < and == have precedence over or

Source: Zybooks, Table 4.7.1

Top Hat Question # 13

What is the output?

```
integer = 4
if(integer > 4 + 2):
    print('integer is greater than 4')
    integer = 4 + 2
else:
    print('integer is not greater than 4. The integer value is', integer, '.')
```

Code blocks and indentation

- A code block in Python is defined by its indentation level

- Highly recommended to use 4 spaces per indentation level

- Either use spaces or tabs for indentation (never both!). It's very likely you will end up with an IndentationError if you use both!

Code blocks and indentation

- A code block in Python is defined by its indentation level

- Highly recommended to use 4 spaces per indentation level

- Either use spaces or tabs for indentation (never both!). It's very likely you will end up with an IndentationError if you use both!

```
prices = [100, 55, 66, 75, 99]
return_value = 99 in prices

if(return_value == True)
    print('the index of element 99 in list is', list.index[99])
else:
    print('Element 99 is not in list')
    print('Try another number')
```

Code blocks and indentation

- A code block in Python is defined by its indentation level

- Highly recommended to use 4 spaces per indentation level

- Either use spaces or tabs for indentation (never both!). It's very likely you will end up with an IndentationError if you use both!

```
prices = [100, 55, 66, 75, 99]
return_value = 99 in prices

if(return_value == True)
    print('the index of element 99 in list is', list.index[99])
else:
    print('Element 99 is not in list')
    print('Try another number')
```

- Not more than 80 columns or 120 columns of text. Split code on multiple lines if necessary

Top Hat Question # 14

What is the output if input is 33?

```
age = int(input())

if(age ==18):
   print('ready for college?')
elif(age == 22):
   print('ready for grad school?')
else:
   if(age < 18):
      print('too young')
   else:
   print('you are probably in grad school or already have a job')
```

References: accessed June 12, 2019

- **Programming languages types**
  Source: https://www.youtube.com/watch?v=1OukpDfsuXE

- **Eclipse logo**
  Source: https://www.eclipse.org/

- **PyDev logo**
  Source: https://www.pydev.org/

- **Garbage collector picture**
  Source: www.zybooks.com