

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

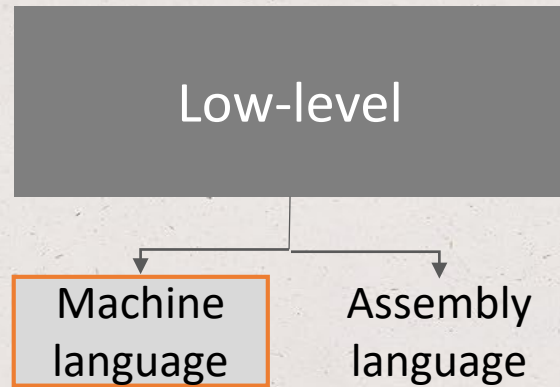
Created in 1940s

- '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

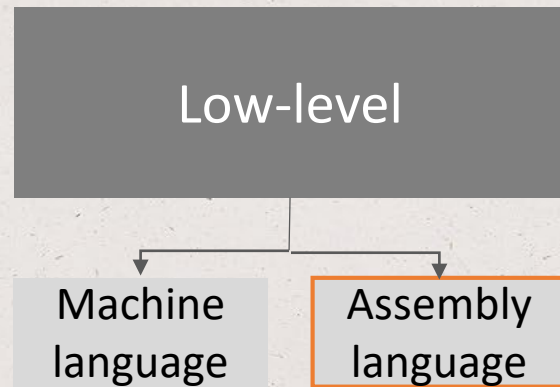


- The only language directly understood by a computer; does not need to be translated (by a compiler or interpreter – more on this later)
- All instructions use binary notations and are written as strings of 1s and 0s

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

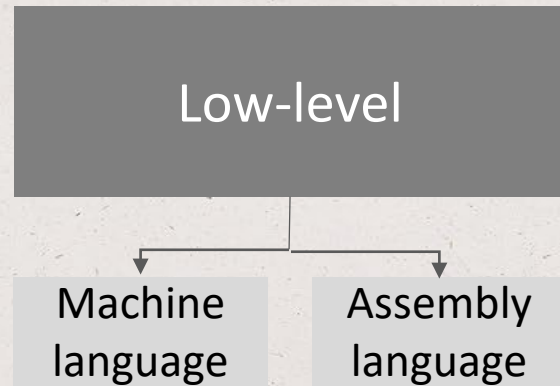


- Consists of a set of symbols and letters
- Requires an assembler to translate the assembly language to machine language
- A second generation because it no longer requires a set of 1s and 0s to write instructions, but terms like:

```
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



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

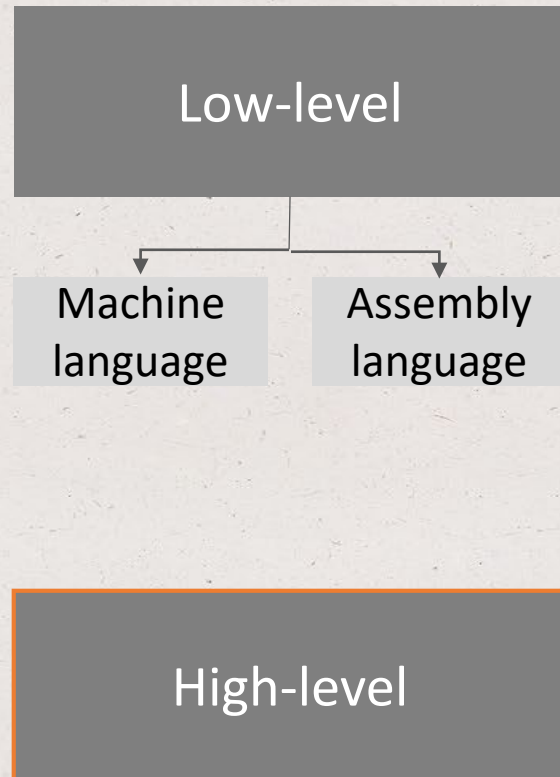
numbers = range(1, 50)
chosen = []

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

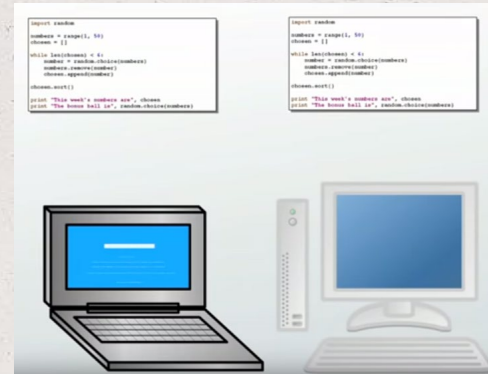
chosen.sort()

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

Programming languages - Types

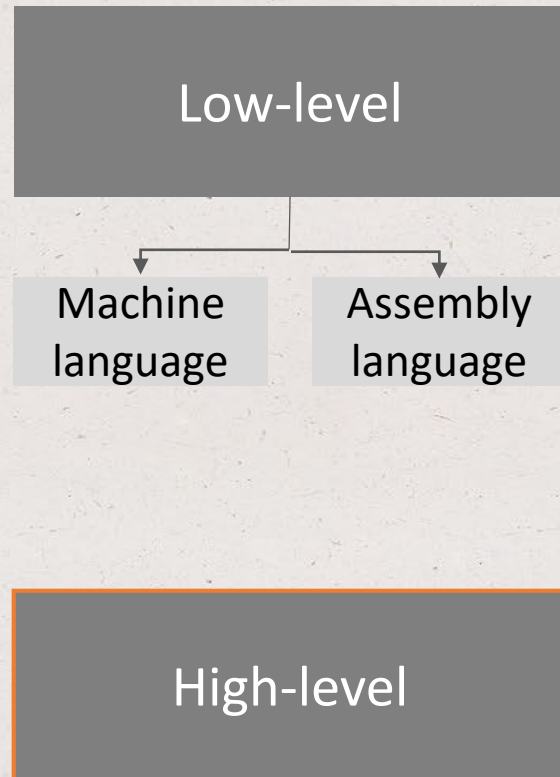


- To learn to program in a high-level language you need to learn *commands*, *syntax*, and *logic*, which correspond closely to vocabulary and grammar
- The 'high-level' program is often portable, the same program can run on different hardware (not necessary OP)

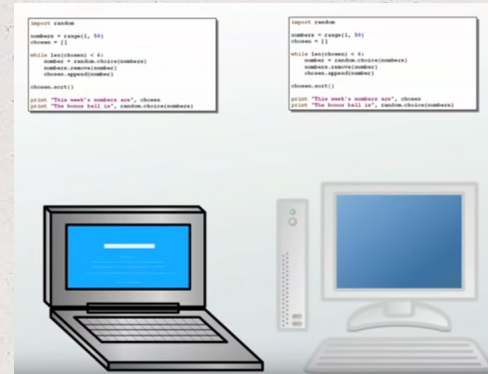


- [Both machine language and assembly language are machine specific, and not portable (the machine code has to be modified to run in another computer)]

Programming languages - Types

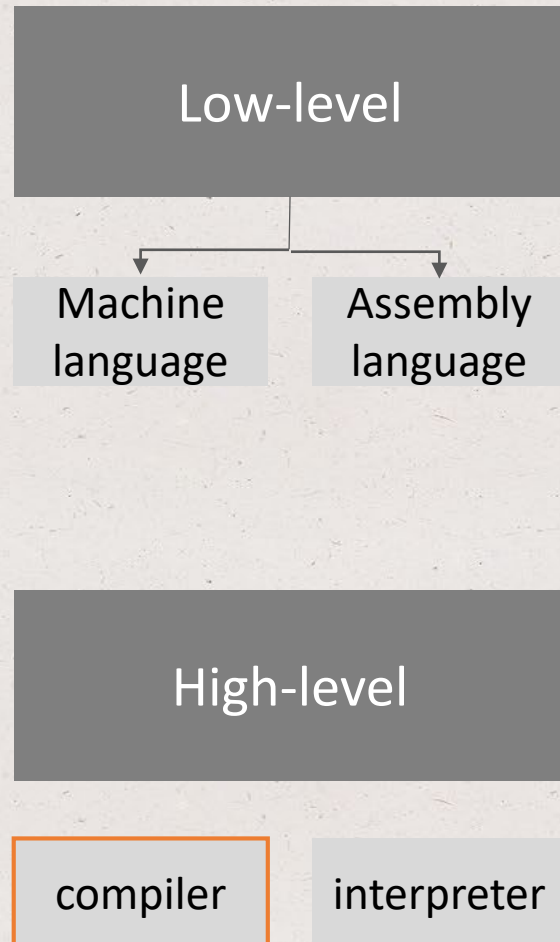


- To learn to program in a high-level language you need to learn *commands*, *syntax*, and *logic*, which correspond closely to vocabulary and grammar
- The 'high-level' program is often portable, the same program can run on different hardware (not necessary OP)

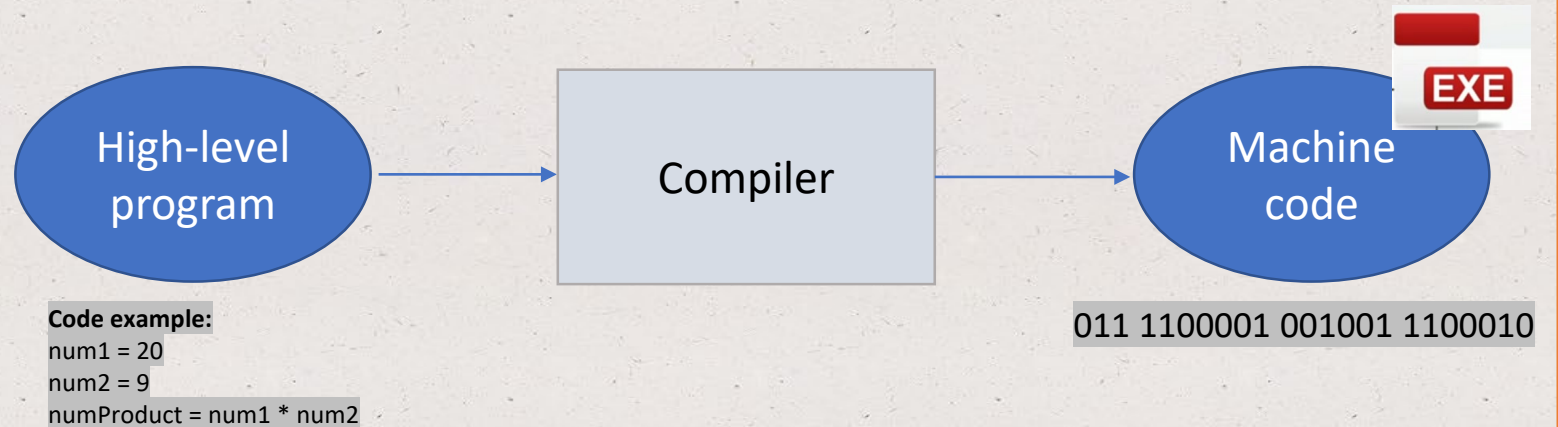


- The code cannot be directly understood by a computer, it needs to be *translated into machine code* using a **compiler** or **interpreter**

Programming languages - Types

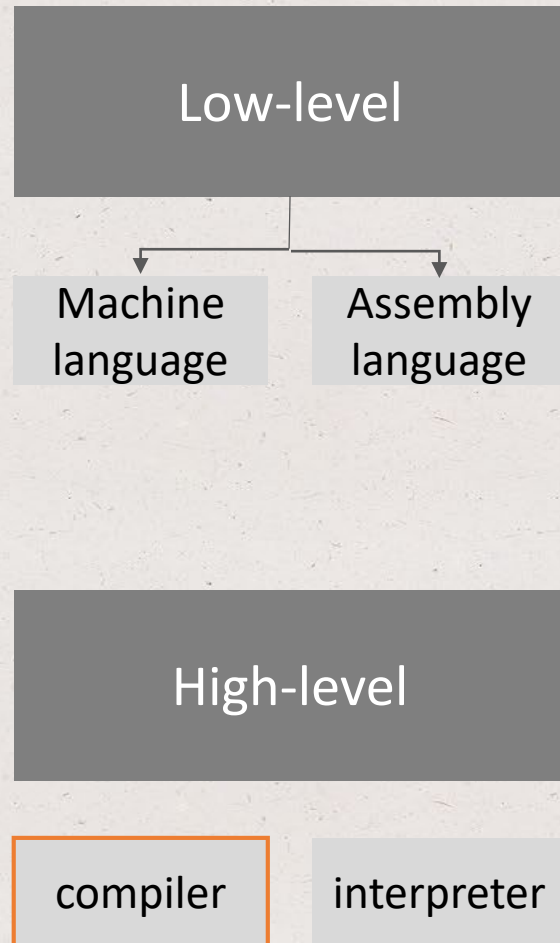


- The compiler is used to translate a high-level program into machine code (the result is an .exe file)

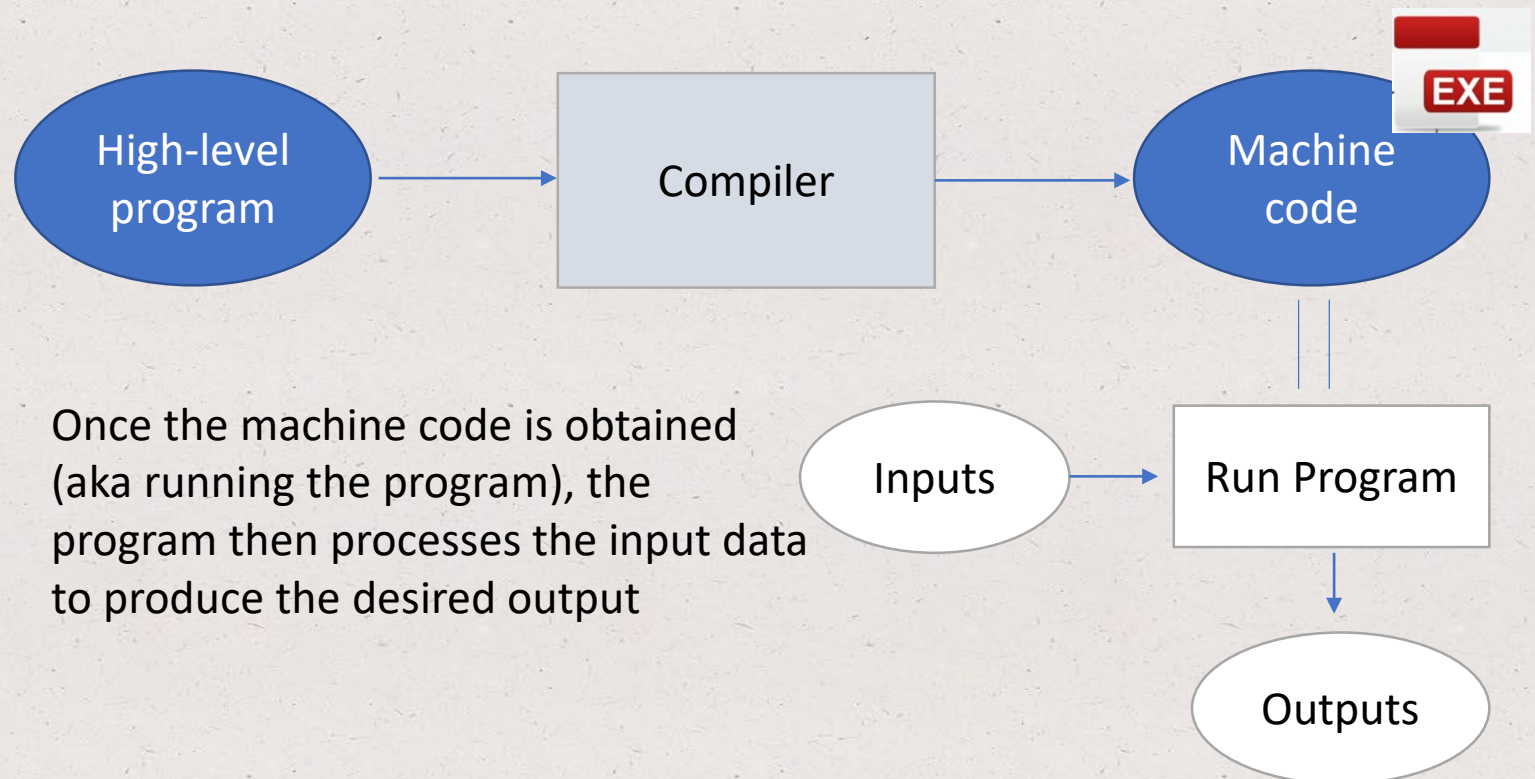


- Once you have an executable (.exe) file you can run the program over and over again without having to compile it again (e.g. you only need to translate (compile) a *movie* only once and can be used many times afterwards)

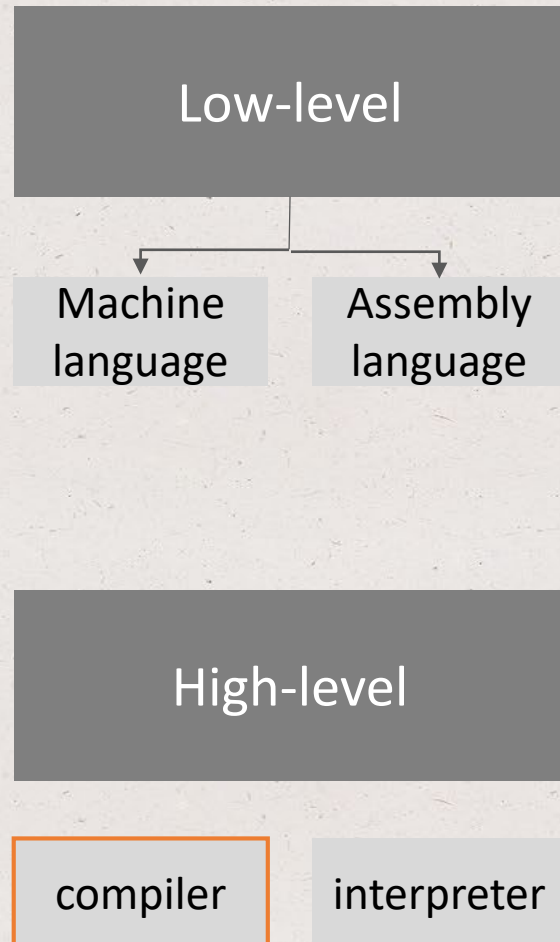
Programming languages - Types



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



Programming languages - Types



- **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)

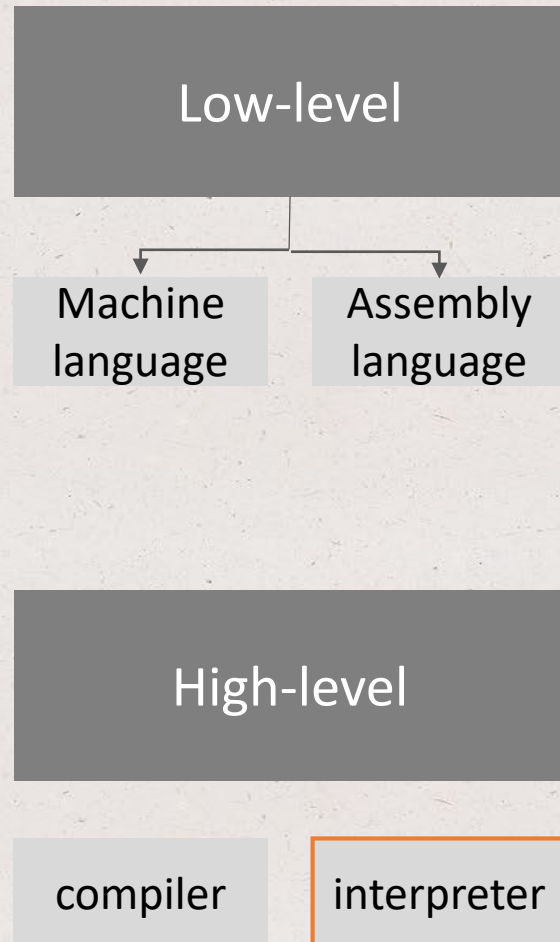


- **Disadvantages:**

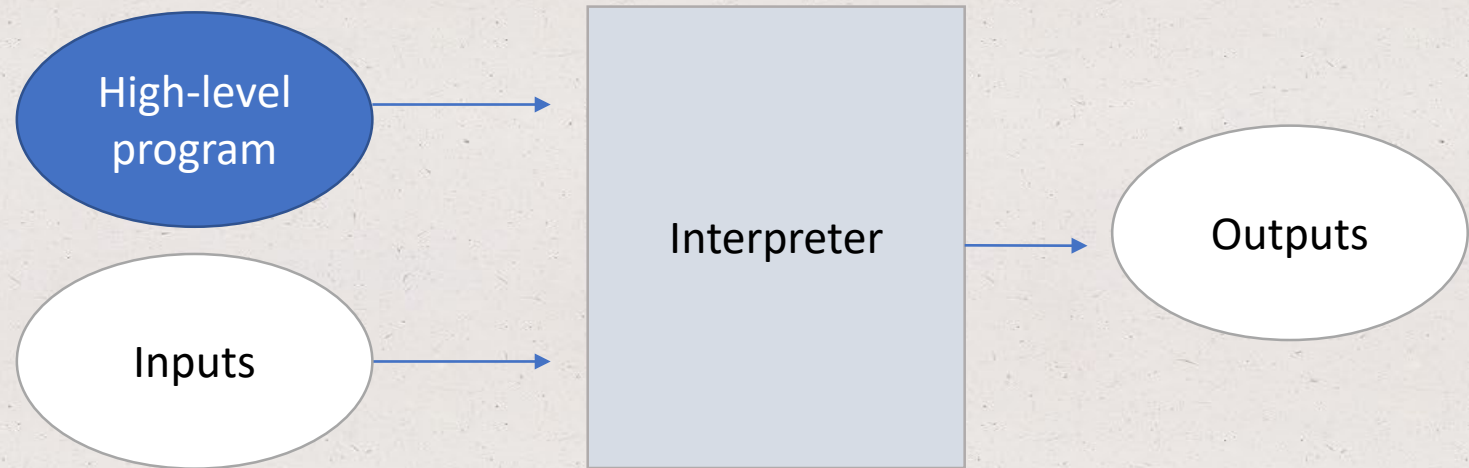
- > slower than interpreter languages

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

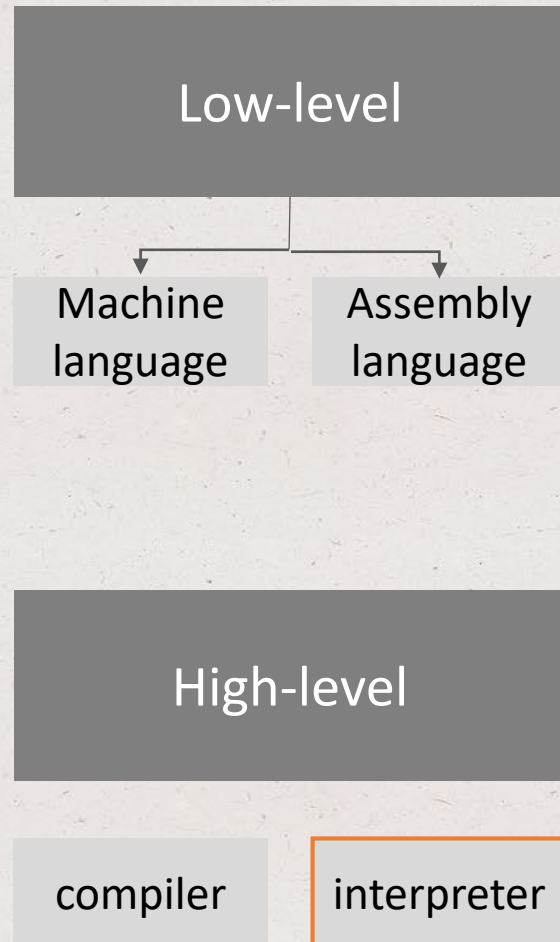
Programming languages - Types



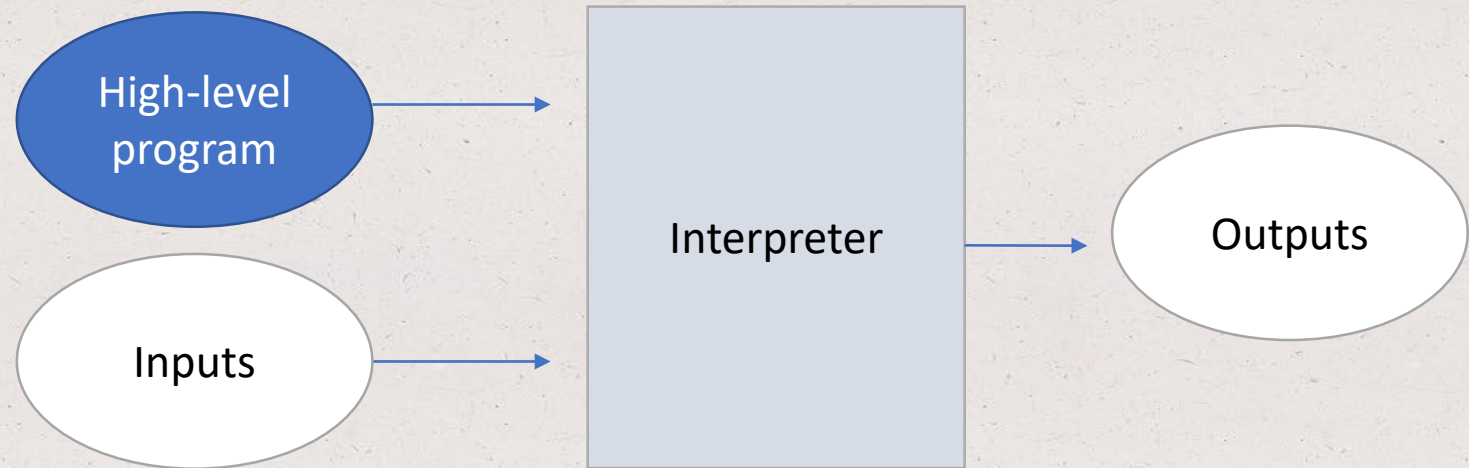
- An interpreter is a computer program that *simulates a computer that understands a high-level language*
- The interpreter translates the high-level program line by line during execution, which results in the desired output data
- The only result is the output data, there is **no compile code**



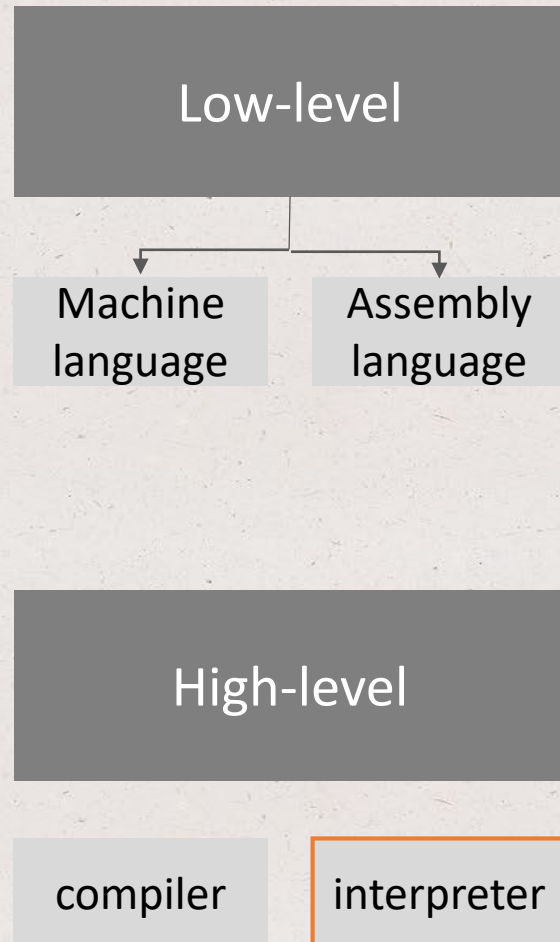
Programming languages - Types



- 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)



Programming languages - Types



- **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

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. <code>print("hello world")</code> in Java and Python)	overlaps (e.g. <code>imprimer</code> and <code>imprimir</code> are verbs for "print" in French and Spanish)
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

P [edit]			
• P	• Picolisp	• PL360	• PPL
• P4	• Pict	• PLANC	• Processing
• P''	• Pig (programming tool)	• Plankalkül	• Processing.js
• ParaSail (programming language)	• Pike	• Planner	• Prograph
• PARI/GP	• PIKT	• PLEX	• PROIV
• Pascal – ISO 7185	• PILOT	• PLEXIL	• Prolog
• PCASTL	• Pipelines	• Plus	• PROMAL
• PCF	• Pizza	• POP-11	• Promela
• PEARL	• PL-11	• POP-2	• PROSE modeling language
• PeopleCode	• PL/0	• PostScript	• PROTEL
• Perl	• PL/B	• Portable	• ProvideX
• PDL	• PL/C	• POV-Ray SDL	• Pro*C
• Perl 6	• PL/I – ISO 6160	• Powerhouse	• Pure
• Pharo	• PL/M	• PowerBuilder – 4GL GUI application generator from Sybase	• PureBasic
• PHP	• PL/P	• PowerShell	• Pure Data
• Pico	• PL/SQL		• Python

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 

[Source:](#)

https://en.wikipedia.org/wiki/Guido_van_Rossum

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 

Source:

https://en.wikipedia.org/wiki/Guido_van_Rossum

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!)

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

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

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 by 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 */

tbasp_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 = tbasp_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>

The screenshot displays the Python Tutor interface for Python 3.6. The code editor on the left contains the following code:

```
1 # define variables
2 x = 2
3 y = 5
4
5 # print sum
6 print(x+y)
```

A green arrow points to line 6, indicating it is the current line of execution. A red arrow points to line 7, indicating the next line to execute. Below the code editor, there is a legend: a green arrow for "line that has just executed" and a red arrow for "next line to execute". A message states: "Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there." Below this is a progress bar and a set of navigation buttons: "<< First", "< Back", "Program terminated", "Forward >", and "Last >>". At the bottom, there is a prompt: "Help improve this tool by clicking whenever you learn something:" followed by two buttons: "I just cleared up a misunderstanding!" and "I just fixed a bug in my code!".

On the right side of the interface, the "Print output (drag lower right corner to resize)" window shows the output "7". Below this, the "Frames" and "Objects" panels are visible. The "Frames" panel shows the "Global frame" with variables "x" and "y" having values "2" and "5" respectively.

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 = ' ')\nprint('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')
```

no space


20's

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: `IndentationError`, `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 # 5

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` \neq `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` \neq `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` \neq `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 # 8

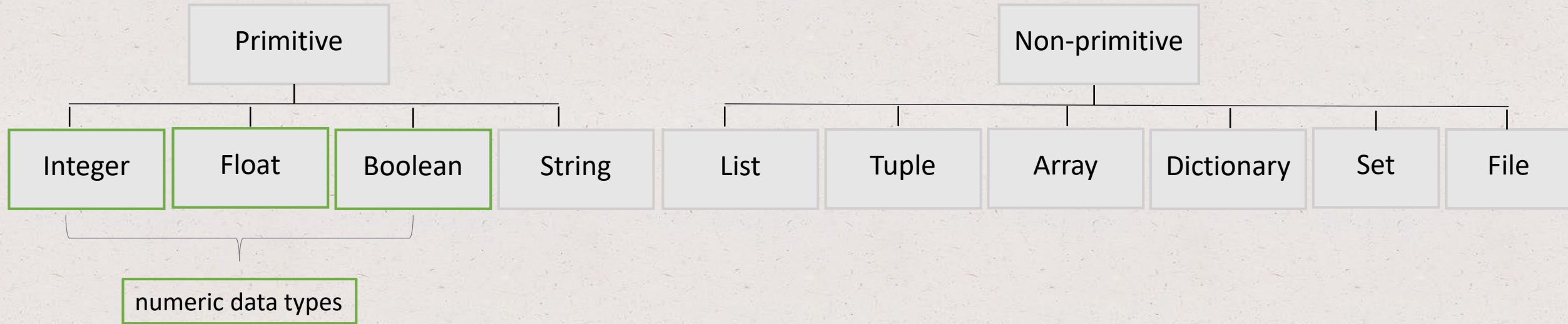
- Write code to swap the values of a and b



Data types



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.

Top Hat Question # 9

What is the result of this expression?

$$(3 \times 2) + (8 + 4) \times 2$$

- (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 remainder of the division of two integer operands



Top Hat Question # 10

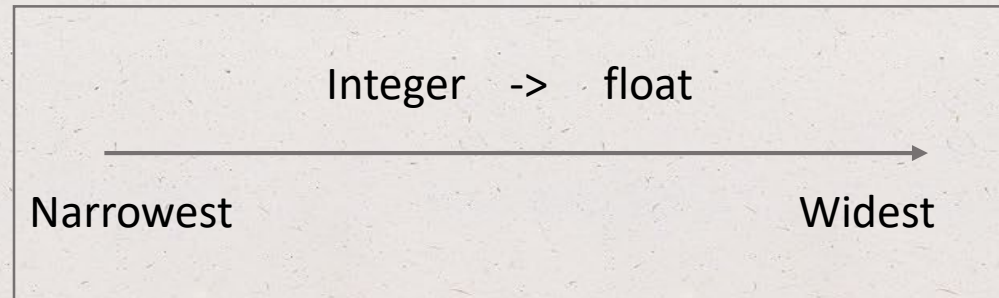
What is the result of these expressions?

(a) $121.0 // 11$

(b) $125/13$

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?



- **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()` converts to float type
 - `str()` converts to string type

Type conversion

- Implicit conversion:

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

data_type

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

- The management of the **Private Heap** is performed by the interpreter itself
- *On demand* allocations by the *Python memory manager* can be done through the Python/C API functions
- Complicated for us, but more info here: <https://docs.python.org/3/c-api/memory.html>

Variable names

Global space

Objects in memory

Heap space

Objects

```
age1 = 18
```

Variable names

Global space

age1

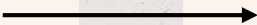
Objects in memory

Heap space

18

98

99



Objects

```
age1 = 18  
age2 = 23.4
```

Variable names

Global space

age2

age1

Objects in memory

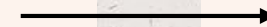
Heap space

23.4

18

98

99



Objects

```
age1 = 18  
age2 = 23.4  
age2 = age1
```

Variable names

Global space

Name binding:

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

age2

age1

Objects in memory

Heap space

23.4

18

98

99



Objects

Garbage collector (deletes objects with no reference to optimize memory)



```
age1 = 18  
age2 = 23.4  
age2 = age1
```

Variable names

Global space

Objects in memory

Heap space

Name binding:

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

age2

23.4

98

age1

18

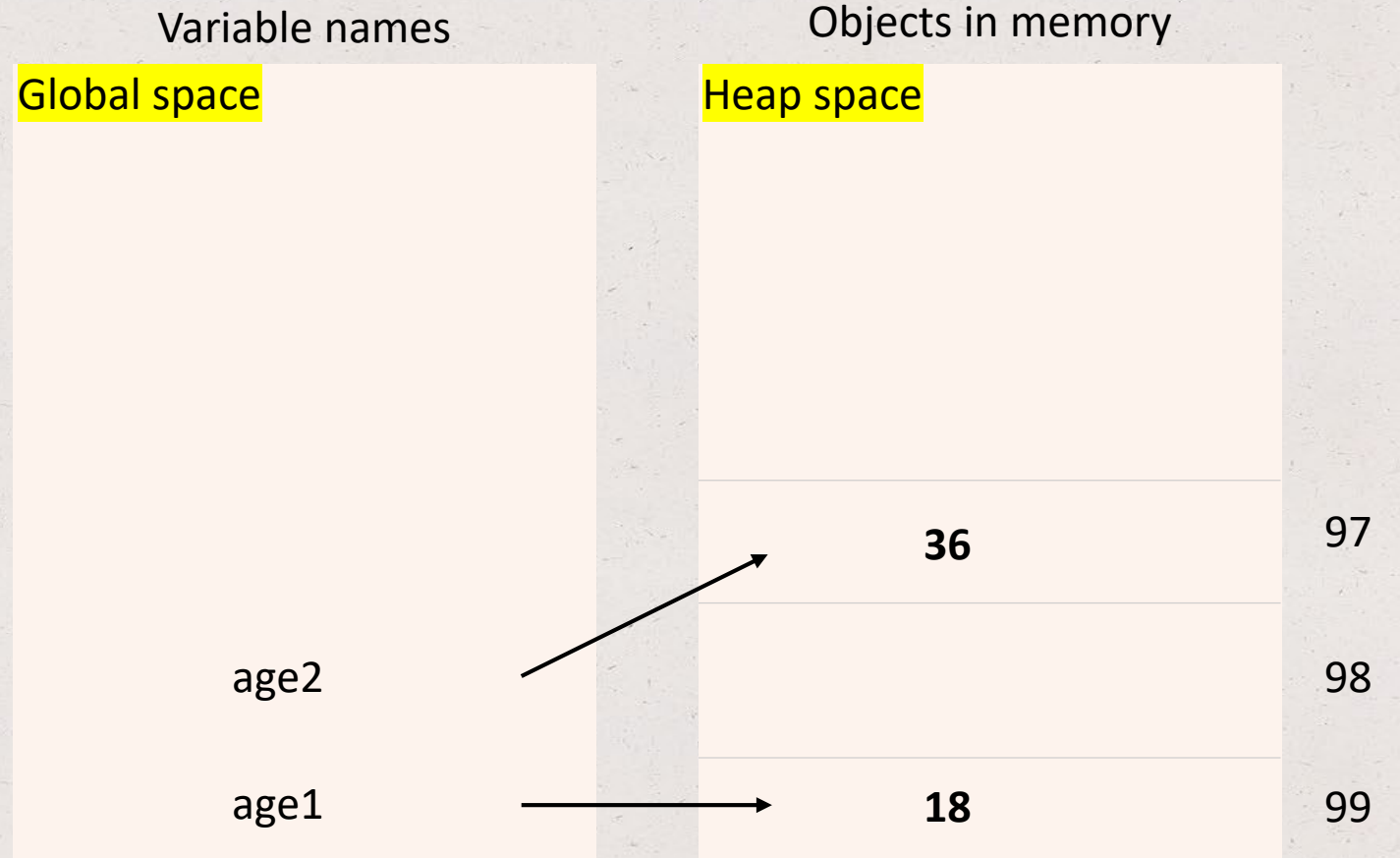
99

Objects

Garbage collector (deletes objects with no reference to optimize memory)



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



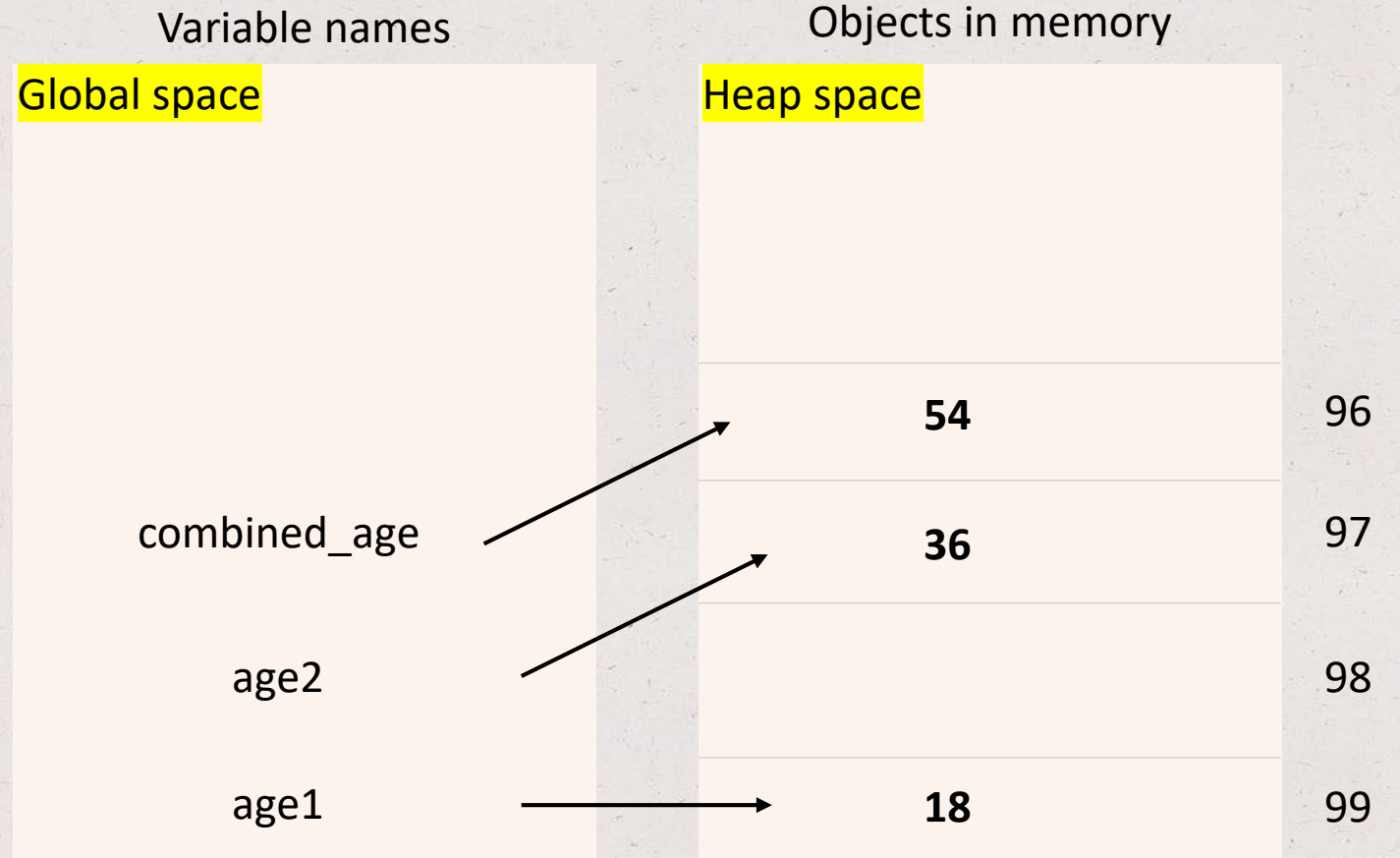
Integers and strings are **immutable** – modifying their value results in a new object being created

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
```



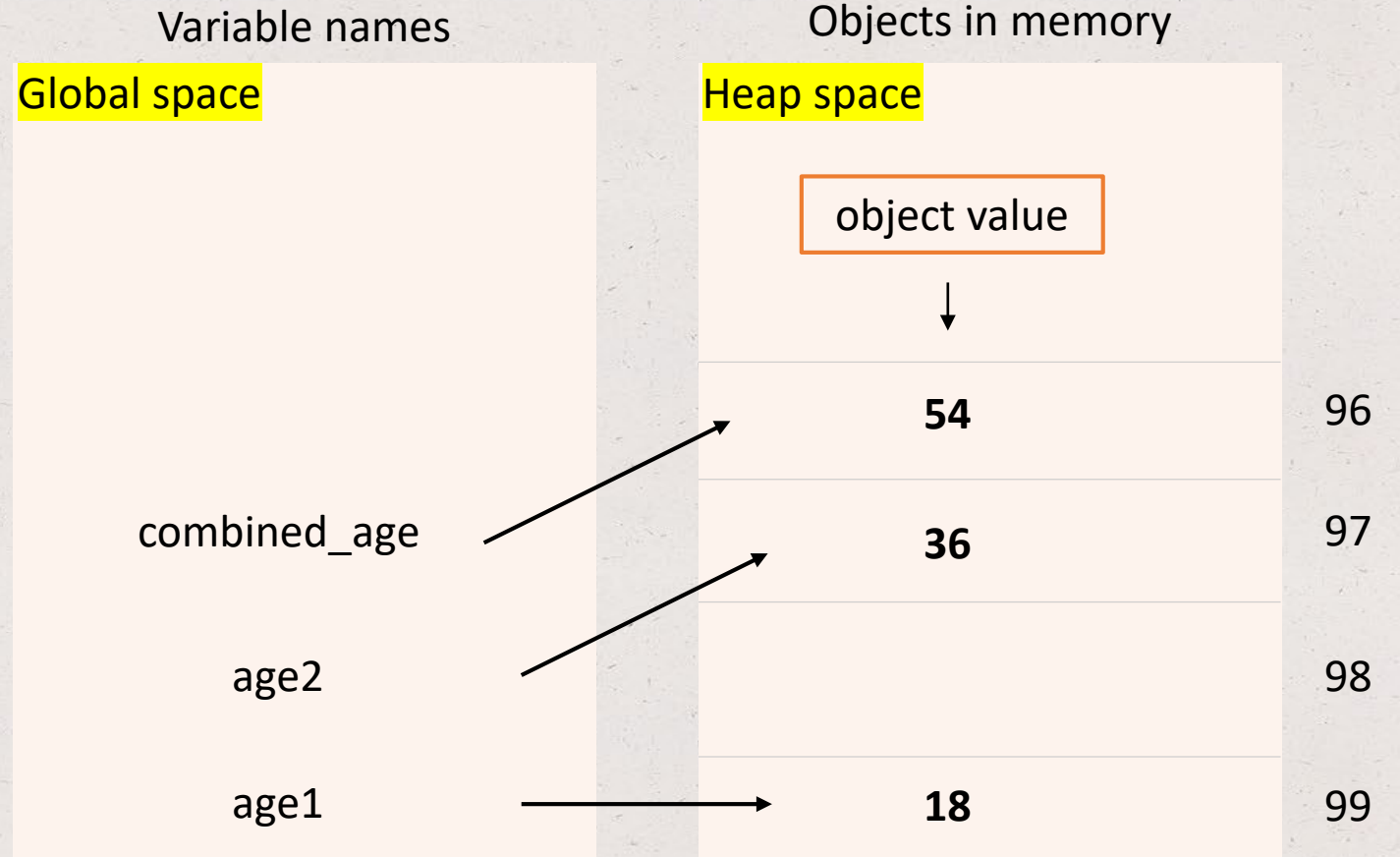
Objects

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)
```



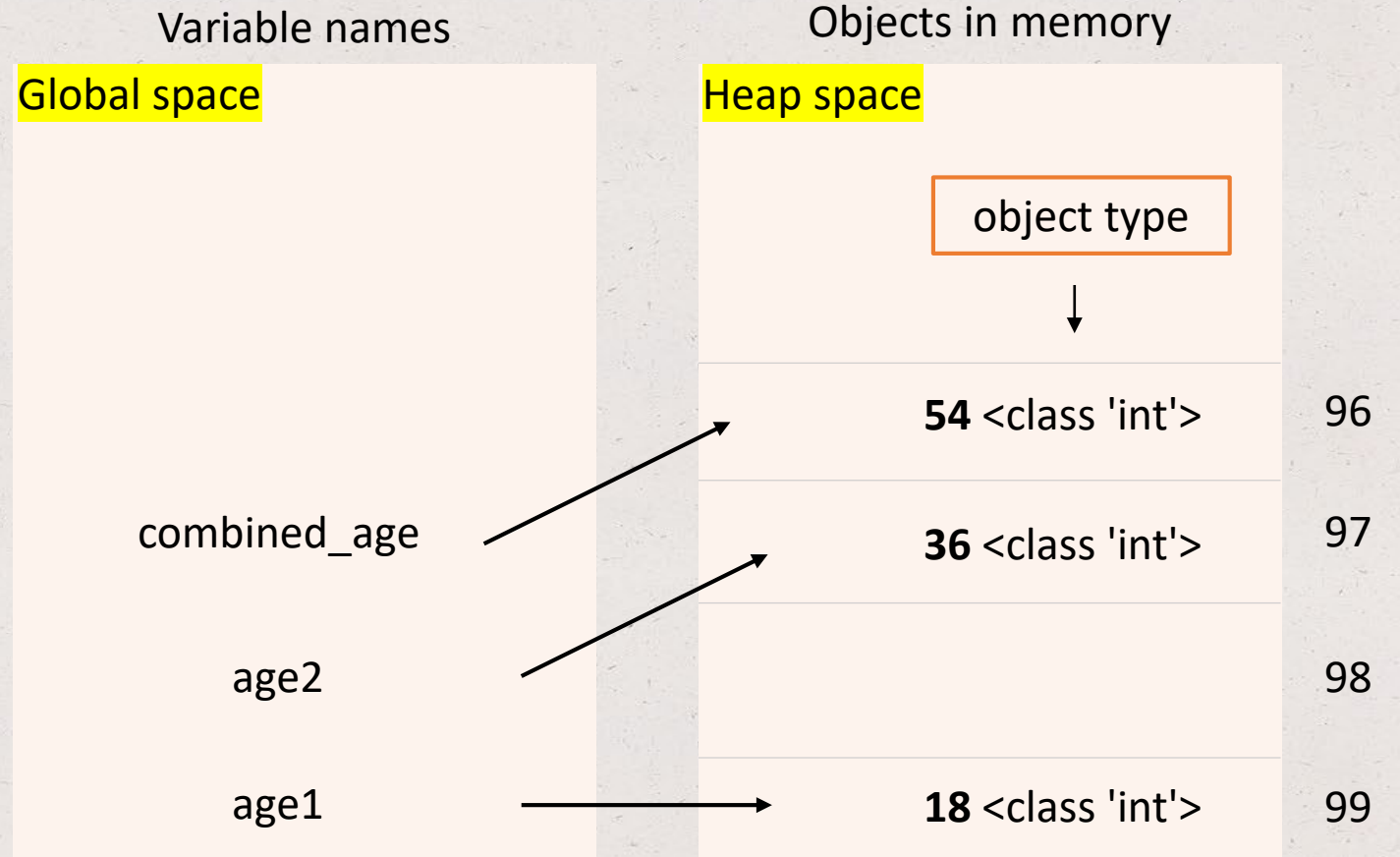
Objects

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 type
print(type(age1))
```



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

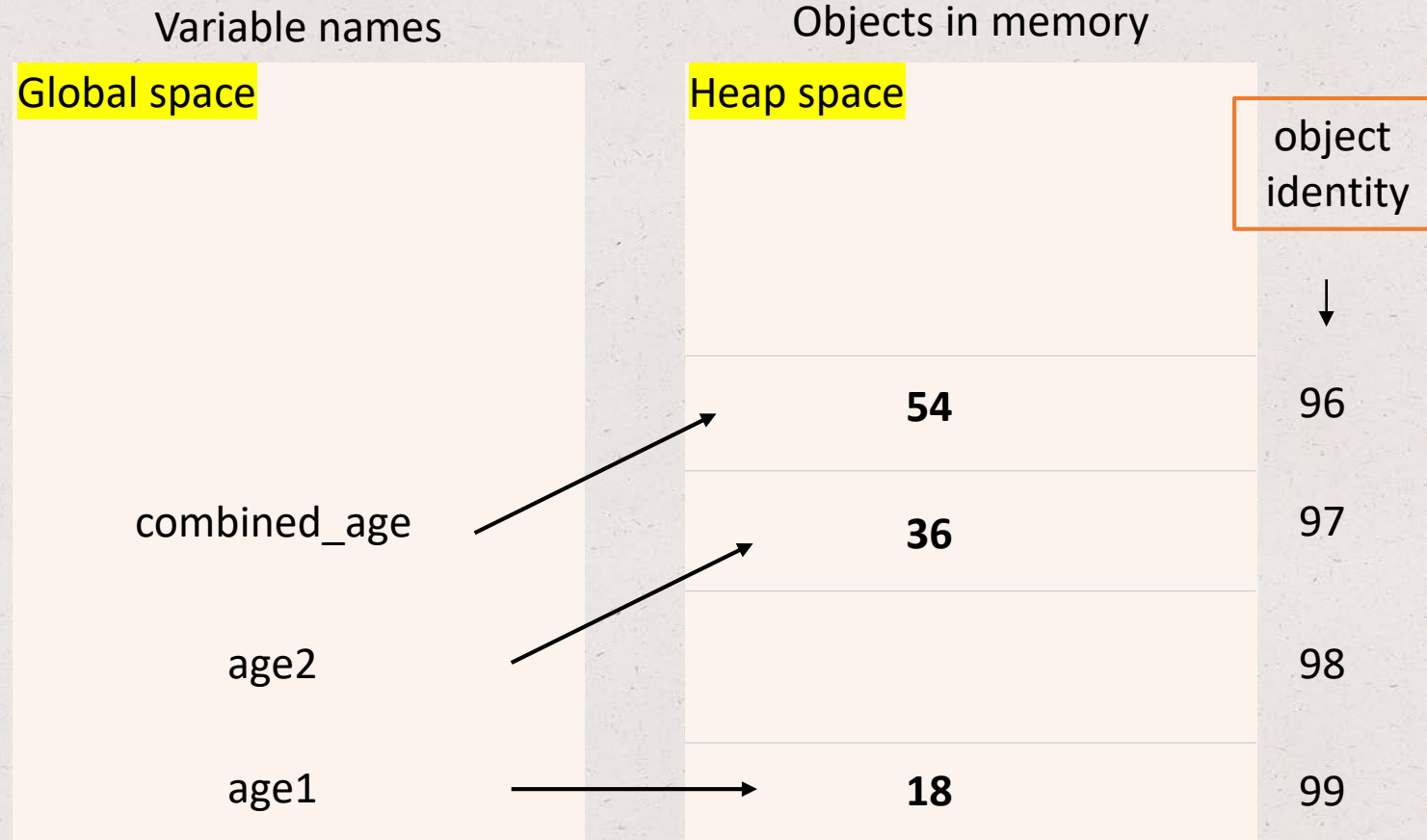
Objects

Garbage collector (deletes objects with no reference to optimize memory)



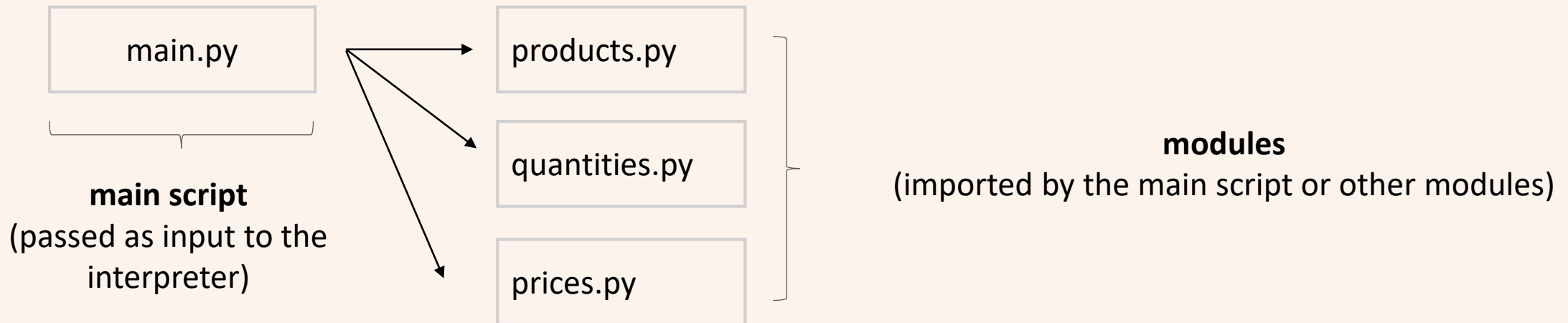
```
age1 = 18
age2 = 23.4
age2 = age1
age2 = age2 * 2
combined_age = age1 + age2
```

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



Module basics

- Good programming practice is to have a:
 - Main script (calls modules)
 - Modules (can also call other modules)
- A module is called by the main script (or other modules) using the **import** command



Module basics

- Good programming practice is to have a:
 - Main script (calls modules)
 - Modules (can also call other modules)
- A module is called by the main script (or other modules) using the **import** command

main.py

```
# list product categories
import products
print('Store A product is:', products.store_A)
print('Store B product is:', products.store_B)
```



products.py

```
store_A = 'Organges'
store_B = 'Apples'
```

Module basics

- Good programming practice is to have a:
 - Main script (calls modules)
 - Modules (can also call other modules)
- A module is called by the main script (or other modules) using the **import** command

main.py

```
# list product categories
import products
print('Store A product is:', products.store_A)
print('Store B product is:', products.store_B)
```

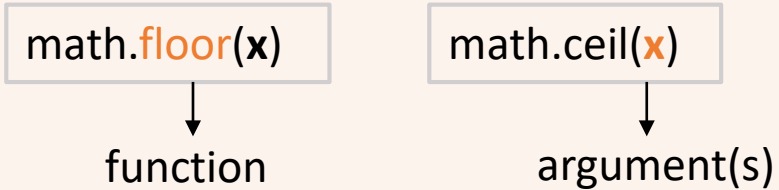


products.py

```
store_A = 'Oranges'
store_B = 'Apples'
# Executes only if run as a main script
if __name__ == '__main__':
    print(' Store A product is', store_A)
```


Module basics

- Python comes with a set of **standard modules** such as *Math*, *Random*, *Datetime*
- Check the Python module index here: <https://docs.python.org/3/py-modindex.html>
- A module is a collection of functions, with one or multiple arguments:



- To use a module (e.g. `math`) simply type: *import math* at the top of the program
- `floor()` and `ceil()` are functions belonging to the module `math`
- A **module is** a Python **object** that you can reference

The math module

```
import math  
float_no1 = float(input())
```

Variable names

Global space

float_no1

Objects in memory

Heap space

3

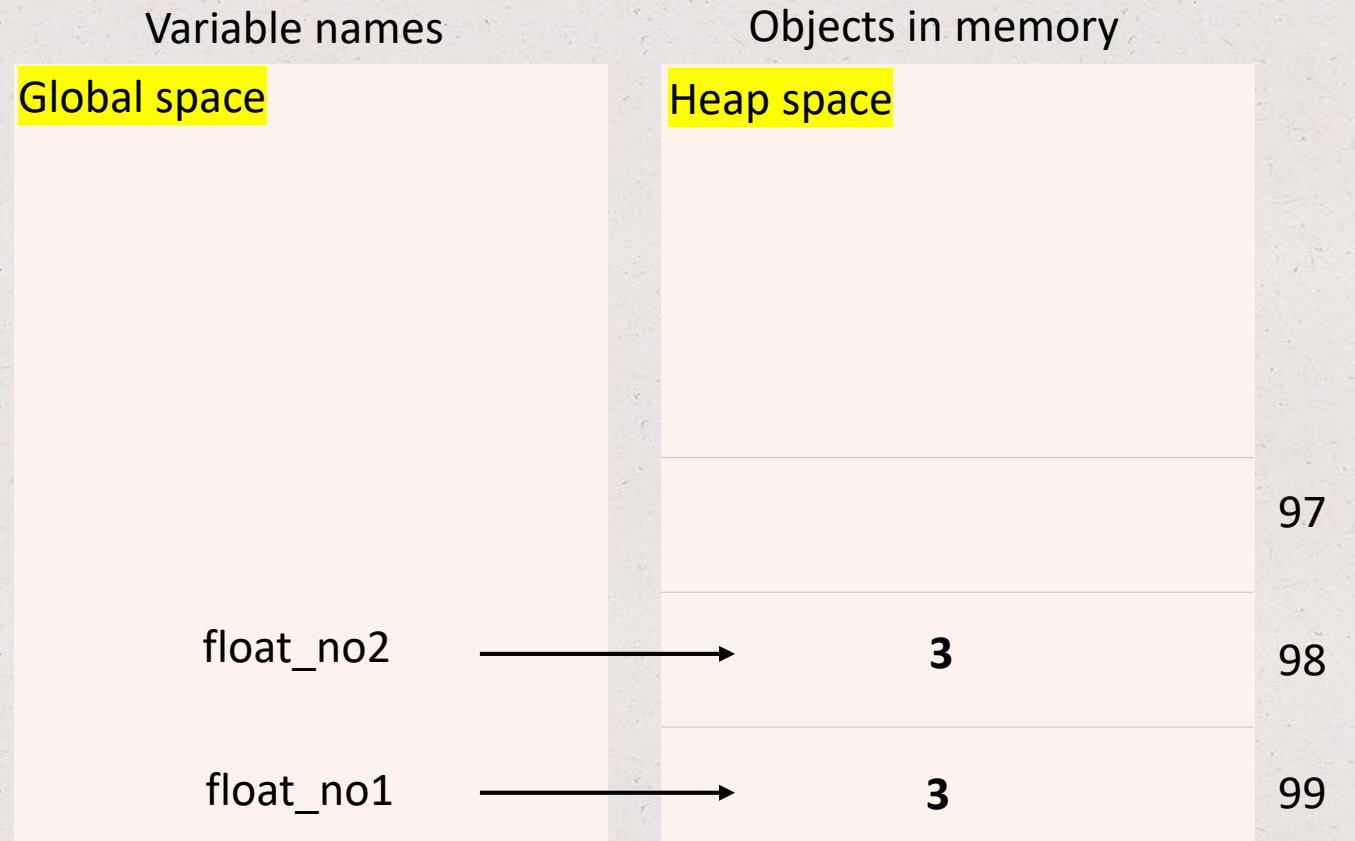
97

98

99

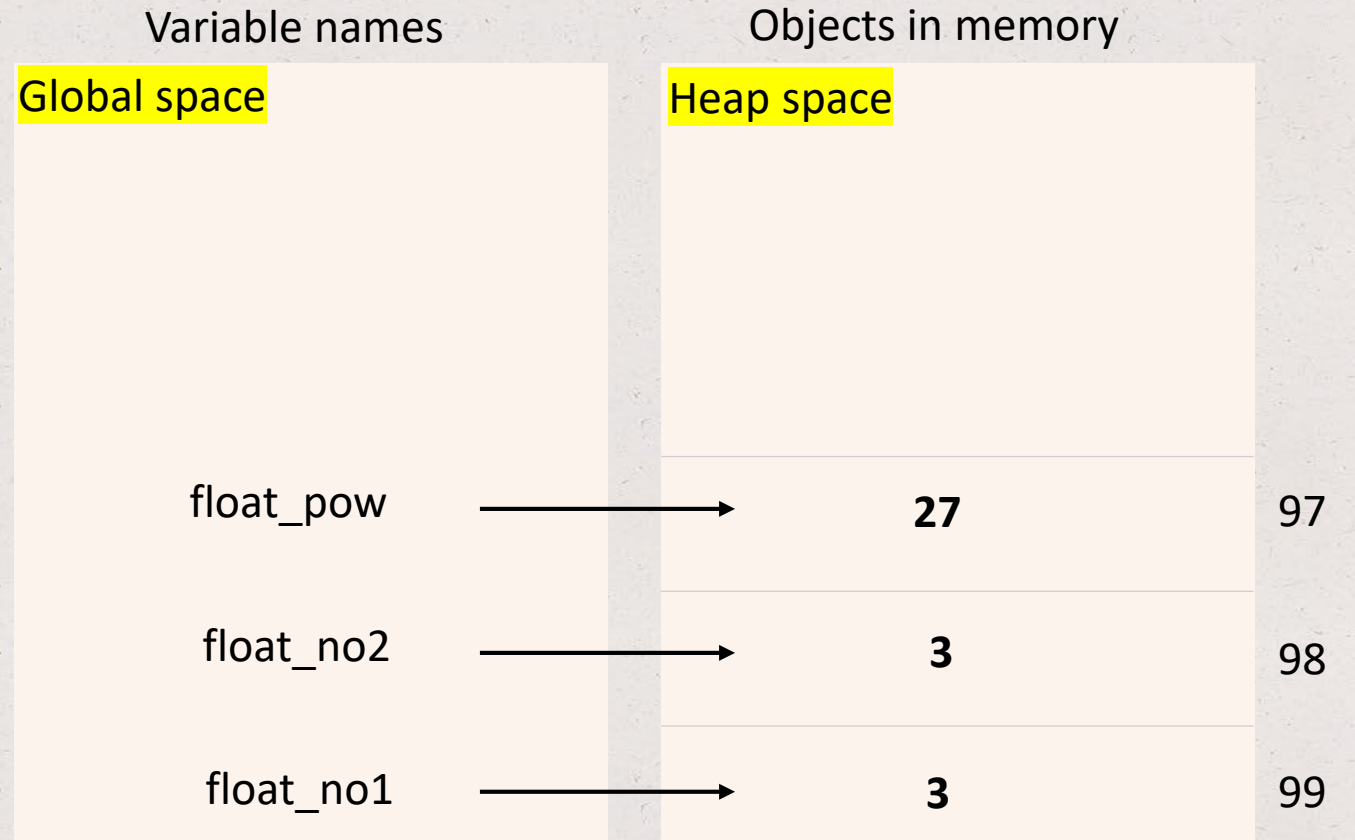
The math module

```
import math
float_no1 = float(input())
float_no2 = float(input())
```



The math module

```
import math
float_no1 = float(input())
float_no2 = float(input())
float_pow = math.pow(float_no1, float_no2)
```

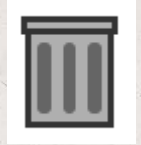


Top Hat Question # 11

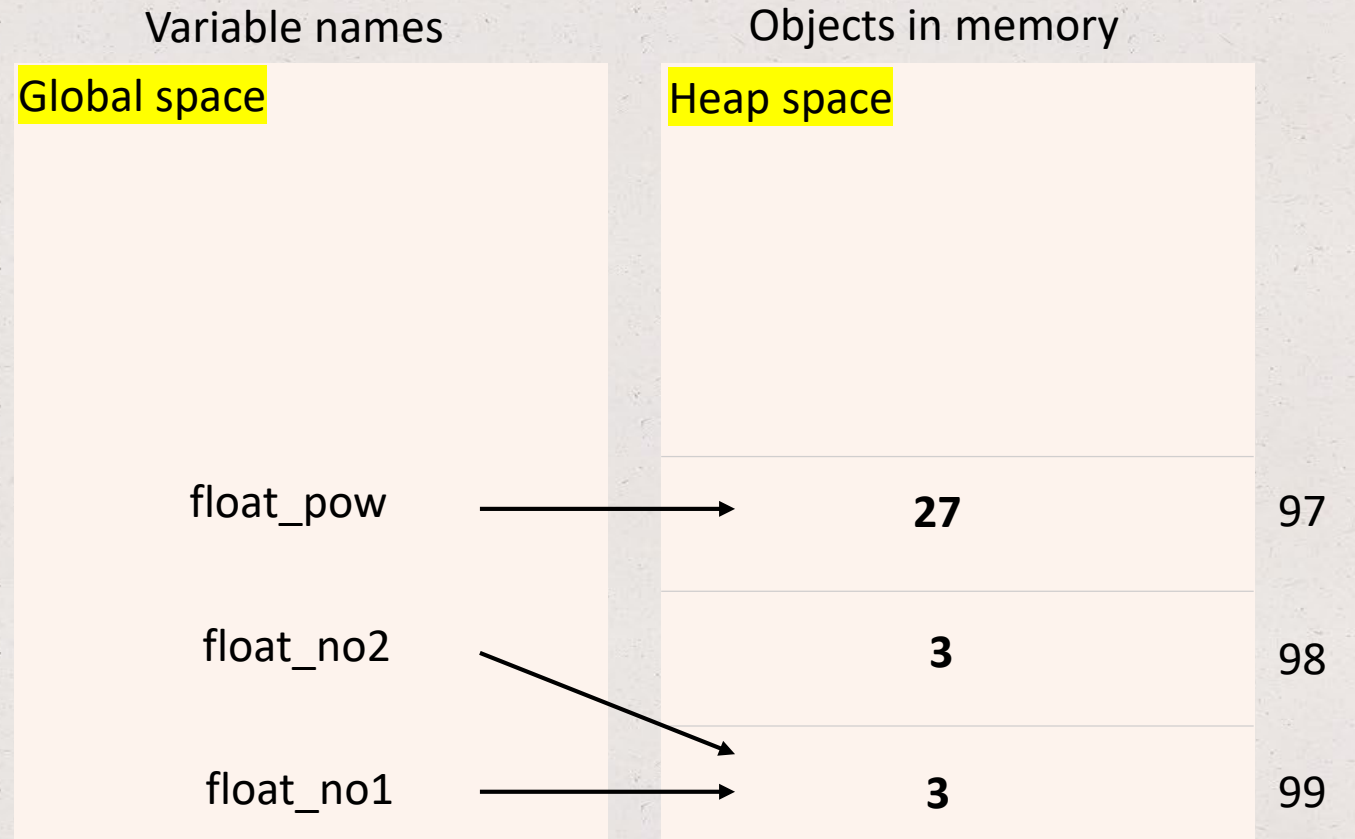
Draw a diagram to explain what happens in the memory of the computer when the following code is run:

```
import math
float_no1 = float(input())
float_no2 = float(input())
float_no2 = float_no1
float_pow = math.pow(float_no1, float_no2)
```

The math module



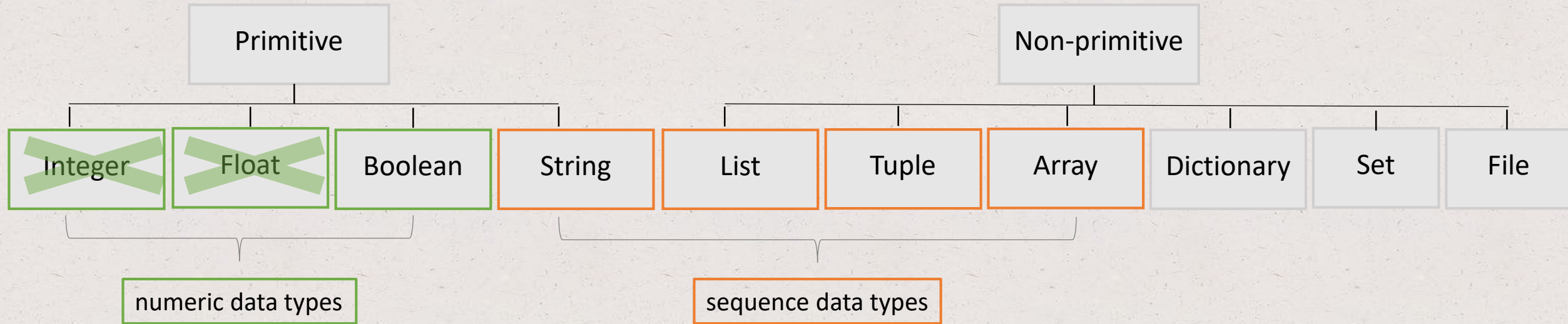
```
import math
float_no1 = float(input())
float_no2 = float(input())
float_no2 = float_no1
float_pow = math.pow(float_no1, float_no2)
```



Chapter 3: Data types

- Sequence data types
- Mapping 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

- Let's talk about **characters** first ...
- **Visible characters:** 'a', 'b', '\$', '#'
- **Escape (special) characters:** `\\` (*backslash*), `\'` (*single quote*), `\''` (*double quote*), `\n` (*new line*), `\t` (*tab, indent*) [once the interpreter reaches the (`\`) character recognizes the start of a special character sequence]
- **Raw string:** place an (**r**) before the string to ignore escape characters

```
print('Today is my \'lucky\' day')
```

```
Today is my 'lucky' day
```

```
print(r'Today is my \'lucky\' day')
```

```
Today is my 'Today is my \'lucky\' day)
```


String basics

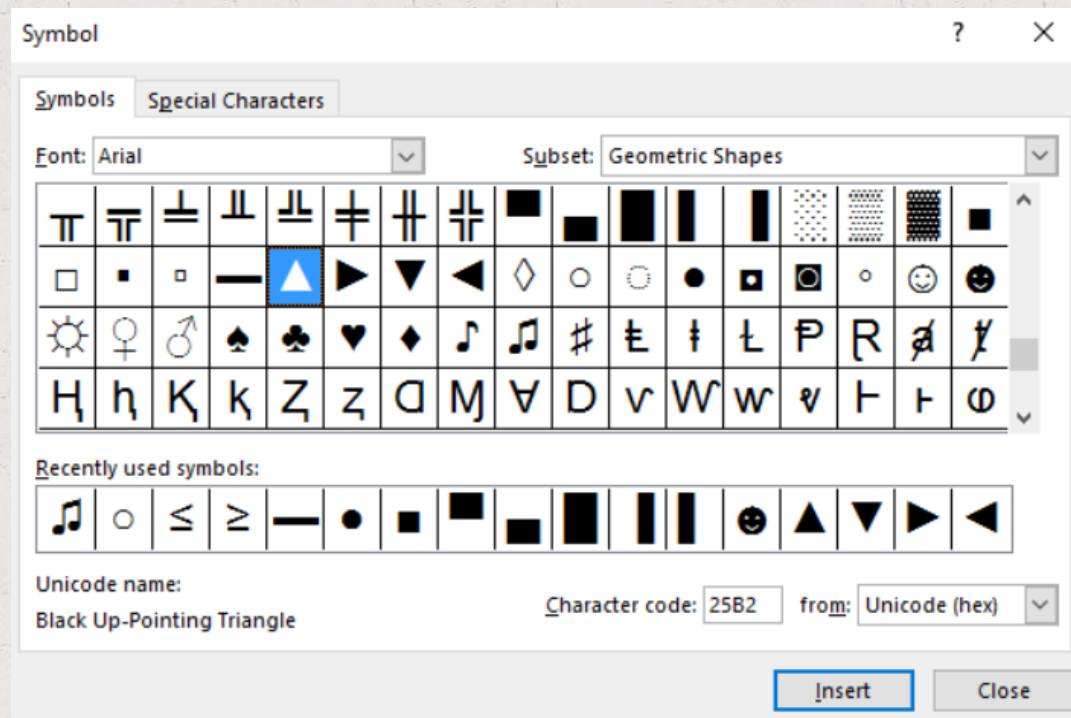
- Computers can only understand machine code (*10010101010101001*)
- Python uses **standards** to represent characters in binary numbers
 - **ASCII table** (uses 7-bits to encode a character, <http://www.asciitable.com/>)
 - **Unicode** (let's you choose between UTF-32, UTF-16, UTF-8 bit encodings, <https://www.fileformat.info/info/charset/UTF-32/list.htm>, <https://www.fileformat.info/info/charset/UTF-16/list.htm>, <https://www.fileformat.info/info/charset/UTF-8/list.htm>)
- Unicode is superior if you have lots of characters to encode (at the expense of larger files...)
- Python 2.x code's default encoding is ASCII
- Python 3.x code's default encoding is UTF-8

String basics

- Computers can only understand machine code (*10010101010101001*)
- Python uses **standards** to represent characters in binary numbers
 - **ASCII table** (uses 7-bits to encode a character, <http://www.asciitable.com/>)
 - **Unicode** (let's you choose between UTF-32, UTF-16, UTF-8 bit encodings, <https://www.fileformat.info/info/charset/UTF-32/list.htm>, <https://www.fileformat.info/info/charset/UTF-16/list.htm>, <https://www.fileformat.info/info/charset/UTF-8/list.htm>)
- Unicode is superior if you have lots of characters to encode (at the expense of larger files...)
- We are not CS guys, why we should care about all these?

String basics

- **Not CS... well... true...** but (just an example) what if you want to write characters that are not supported by a physical keyboard?
- Someone 'magically' created a user friendly interface to imbed special characters in Excel, PP etc.



String basics

- **Not CS... well... true...** but (just an example) what if you want to write characters that are not supported by a physical keyboard?
- Someone 'magically' created a user friendly interface to imbed special characters in Excel, PP etc.
- How do we 'magically' imbed these special characters in Python code?
- Two useful built-in functions:
 - **chr()** to convert an encoded value to a character
 - **ord("")** to convert a character to the encoded value

String basics

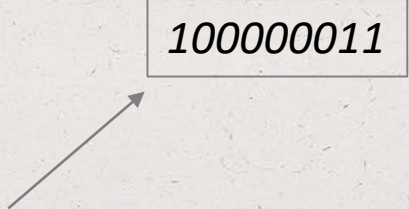
- **Not CS... well... true...** but (just an example) what if you want to write characters that are not supported by a physical keyboard?
- Someone 'magically' created a user friendly interface to imbed special characters in Excel, PP etc
- How do we 'magically' imbed these special characters in Python code?
- Suppose you want to write: 'bună' (means 'hello' in Romanian)

```
print('b' + 'u' + 'n' + chr(259))
```

```
bună
```

String basics

100000011



- 259 is still not a **binary number** (aka base 2 no.). It's actually a **decimal number** (aka base 10 no.)
- The Python interpreter translates this decimal number into binary number (machine code)
- How? see zyBooks Chapter 3.9: 'Binary numbers' for an explanation
- ... enough with characters, let's now see what is a String

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

Global space

my_string

Objects in memory

Heap space

'Hello world'

97

98

99



Top Hat Question # 12

What is the index of 'd' in 'Hello world'?

Top Hat Question # 13

Draw a diagram to explain what happens in the memory of the computer when the following code is run (include type of data in the explanation)

```
my_string = 'Hello world'  
my_number = 5  
my_number = 5 * 2
```


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>

Functions

- is a block of code that is also **called by its name** (independent)
- can have different parameters or may not have at all
- if any **data parameters** are passed, they are **passed explicitly**
- it **may or may not return any data**
- do not deal with Class and its instance concept

Methods

- Is called by its name, but its functions are **associated with an object** (dependent)
- A method is **implicitly passed the object** in which it's invoked
- It **may or may not return any data**
- can operate on the data (instance variables) that is contained by the corresponding class

Top Hat Question # 14

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

Top Hat Question # 15

What is the length of the following string. What is the value at index 3?

```
my_string = 'I need a break!'
```


Top Hat Question # 16

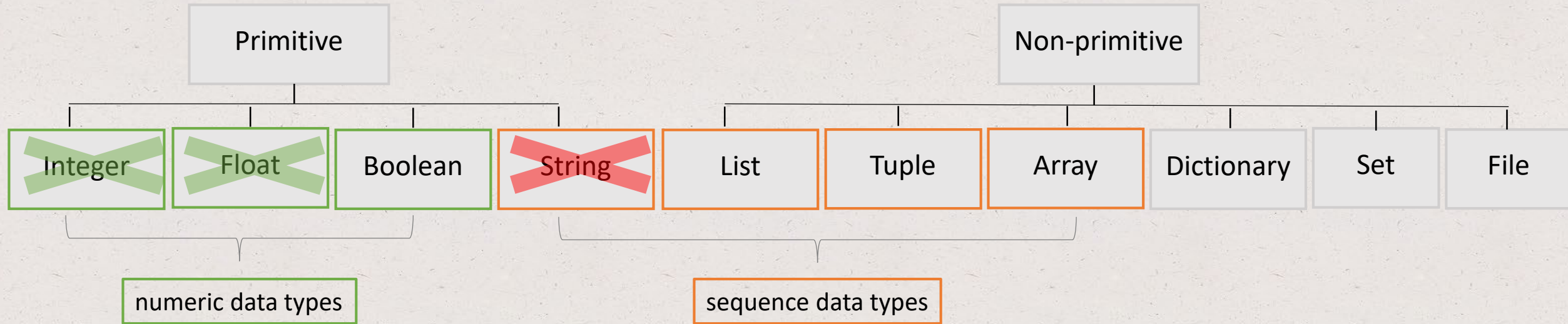
What is the output of the print function? Explain.

```
my_string = 'I need a break!'
my_string[0] = 'W'
my_string[1] = 'e'
print(my_string)
```

String basics – formatting

- How can we augment the ability of the built-in print() function?
- Concepts:
 - conversion **specifier**
 - conversion **operator**
 - conversion **type**
 - Precision
 - Minimum field width
 - '0' conversion flag
 - format()
- **Adam** will cover this topic in the Lab session. More info on zyBooks Ch 7.2 and 7.5

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, or remove elements of a list using built-in methods:
 - `list.append(x)` – adds 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 # 17

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

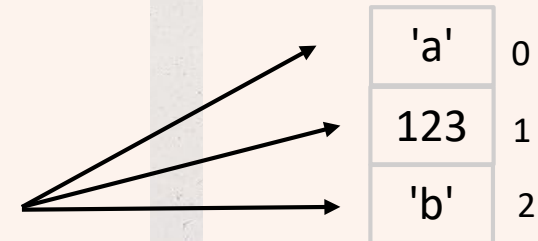
Global space

Objects in memory

Heap space

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

my_list



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:

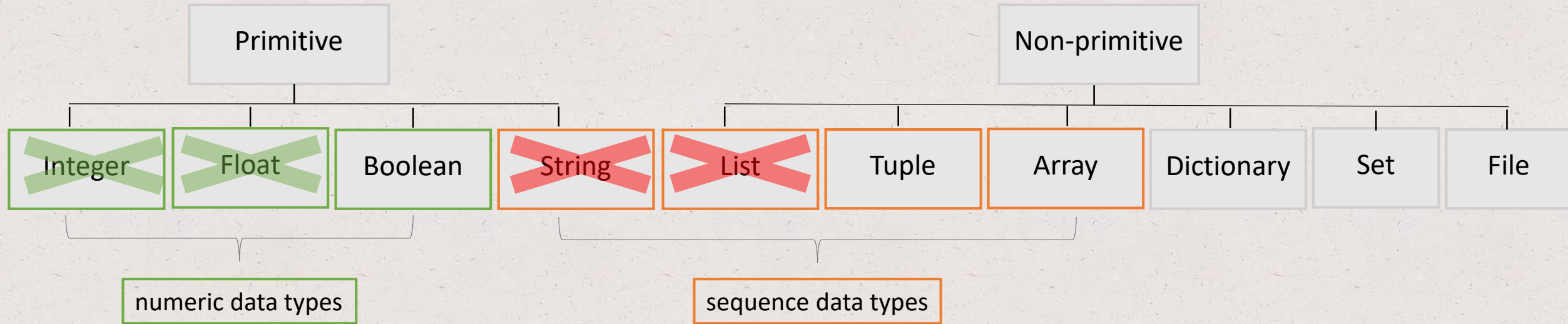
<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Top Hat Question # 18

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 – accessing elements

- Tuple elements can also be **accessed by attribute name**. This type of tuple is a **namedtuple**
- The package** *collections* implements specialized sequence data types providing alternatives to built-in sequence data types (namedtuple is a specialized version of tuple)
- The *namedtuple()* function in the *collection* package assigns meaning to each position in a tuple and allow for more readable, self-documenting code.
- To import the package and the function: **from** collections **import** namedtuple

**A package in Python is a collection of modules and functions

Tuple basics – accessing elements

- Tuple elements can also be **accessed by attribute name**. This type of tuple is a **namedtuple**

```
# import namedtuple from collections package  
from collections import namedtuple
```

```
# generate the named tuple  
Airplane = namedtuple('Airplane', ['make', 'model', 'price']) #price is in million US dollars
```

```
# generate different airplane objects  
boeing_737= Airplane('boeing', 737, 134.9)  
boeing_777= Airplane('boeing', 777, 442.2)
```

```
# list the price of boeing_737  
print(boeing_737.price)
```

More on syntax here:

<https://docs.python.org/2/library/collections.html#collections.namedtuple>

Tuple basics – memory representation

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

Tuple names

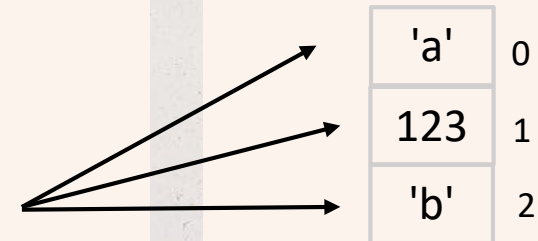
Global space

Objects in memory

Heap space

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

my_tuple



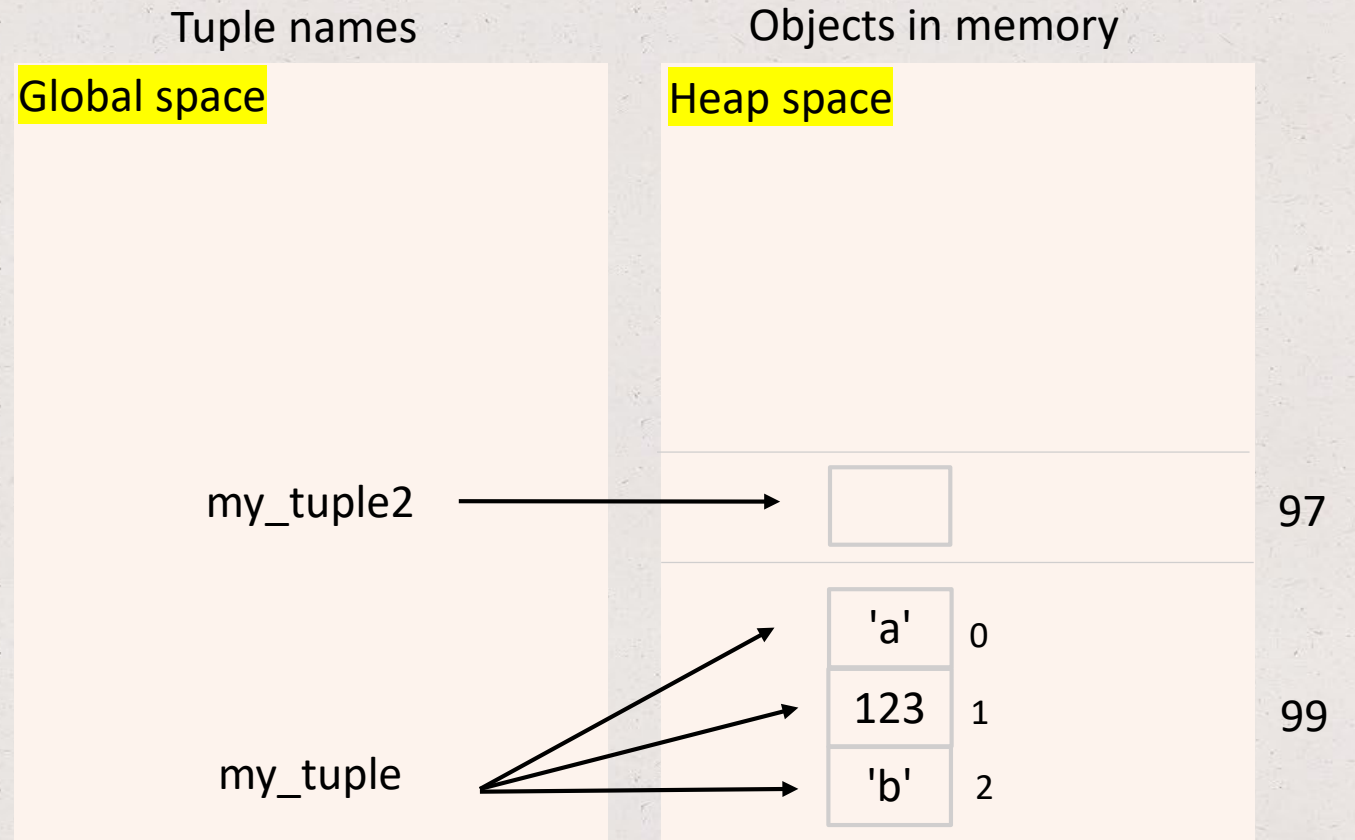
97

99

Arrow meaning: my_tuple holds references to objects in list

Tuple basics – memory representation

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



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

Tuple basics – memory representation

```
# import collections and namedtuple  
from collections import namedtuple
```

```
# generate the named tuple  
Airplane = namedtuple('Airplane', ['make', 'model',  
'price'])
```

Tuple names

Global space

boeing_777

boeing_737

Airplane

Objects in memory

Heap space

98

99

`namedtuple()` only creates the `Airplane` class (more on this later) and its corresponding data types

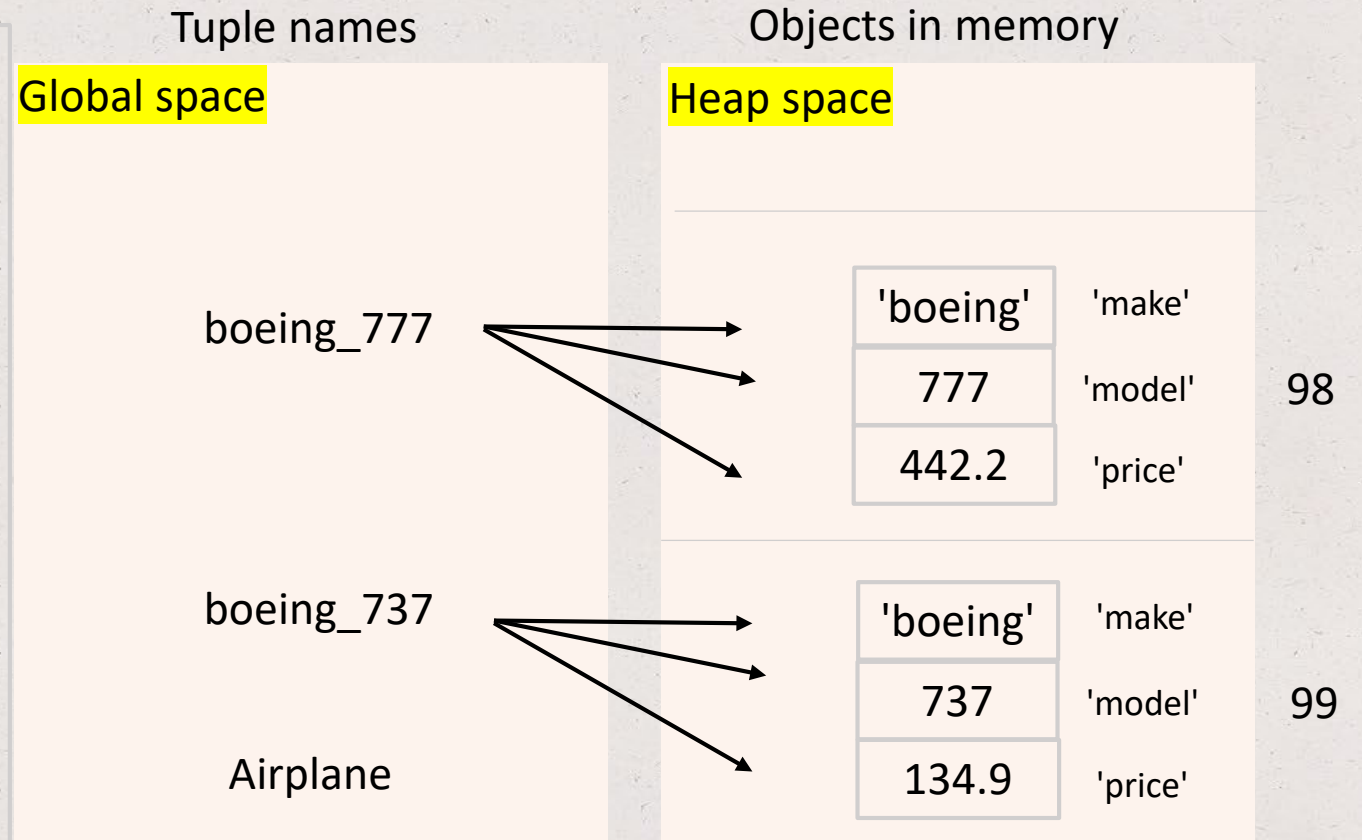
Tuple basics – memory representation

```
# import collections and namedtuple
from collections import namedtuple

# generate the named tuple
Airplane = namedtuple('Airplane', ['make', 'model',
'price'])

# generate different airplane objects
boeing_737= Airplane('boeing', 737, 134.9)
boeing_777= Airplane('boeing', 777, 442.2)

# list the price of boeing_737
print(boeing_737.price)
```



`namedtuple()` only creates the `Airplane` class (more on this later) and its corresponding data types

Top Hat Question # 19

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

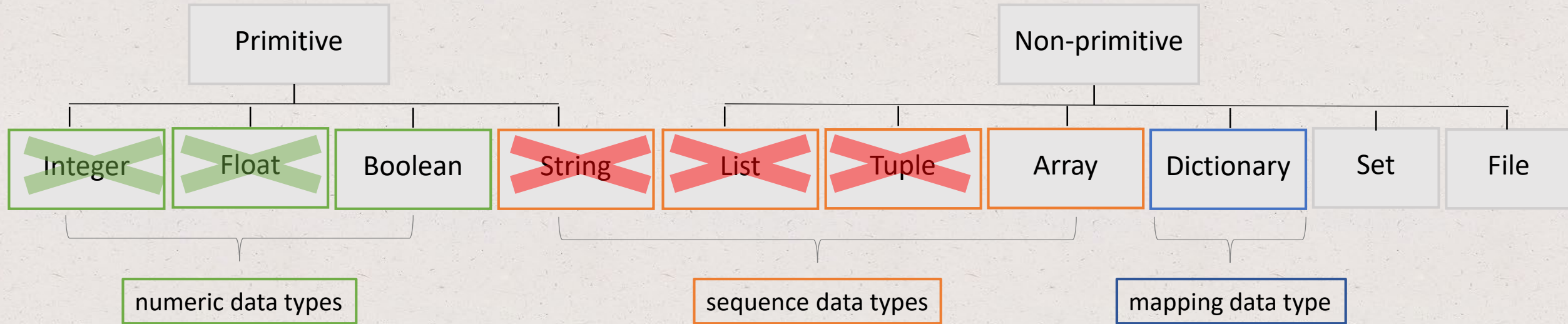
```
# import collections and namedtuple
from collections import namedtuple

# generate the named tuple
Airplane = namedtuple('Airplane', ['make', 'model',
'price'])

# generate different airplane objects
boeing_737= Airplane('boeing', 737, 134.9)
boeing_777= Airplane('boeing', 777, 442.2)

# list the price of boeing_737
print(boeing_737.price)
```

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'
```


Top Hat Question # 20

What is the output?

```
products = {  
    ('GM seed', 'Conv seed'): (120, 140),  
    'Conv' : 87  
}  
print(products['GM seed'])
```

Top Hat Question # 21

What is the output?

```
products = {  
    ('GM seed', 'Conv seed'): (120, 140),  
    'Conv' : 87  
}  
print(products['GM seed', 'Conv 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

airplanes

134.9

'boeing737'

442.2

'boeing777'

98

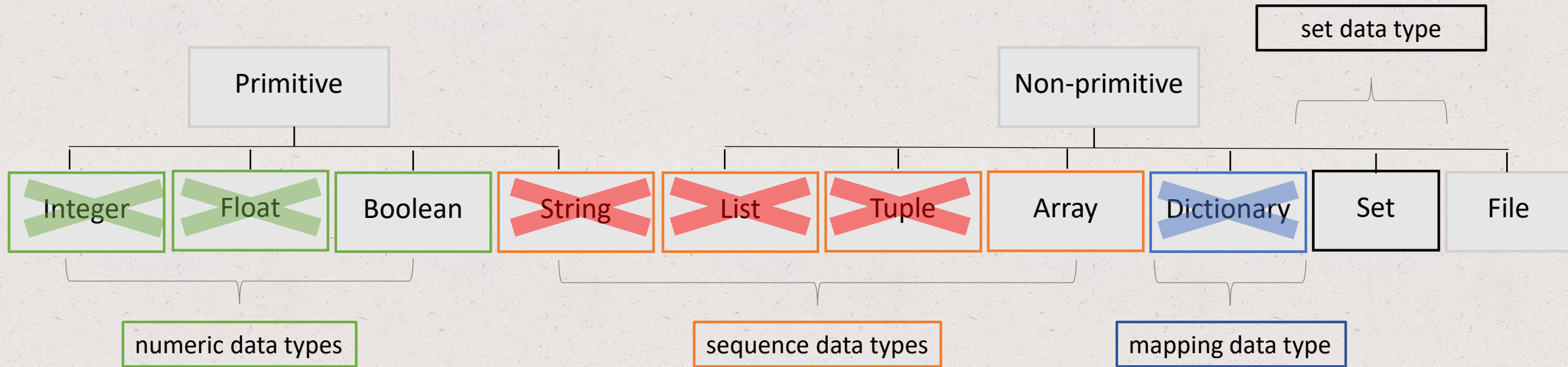
99

Top Hat Question # 22

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 – 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

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

```
{'boeing 737', 'VW Passat', 'VW Tiguan', 'boeing 777',  
'VW Tuareg'}
```


Set basics – memory representation

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

Set names

Global space

Objects in memory

Heap space

airplanes



'boeing 737' 'boeing 777'

98

99

Set basics – memory representation

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

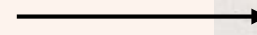
Set names

Global space

Objects in memory

Heap space

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

Global space

Objects in memory

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

Global space

Objects in memory

Heap space

cars



'VW Tiguan' 'VW Passat'

98

airplanes

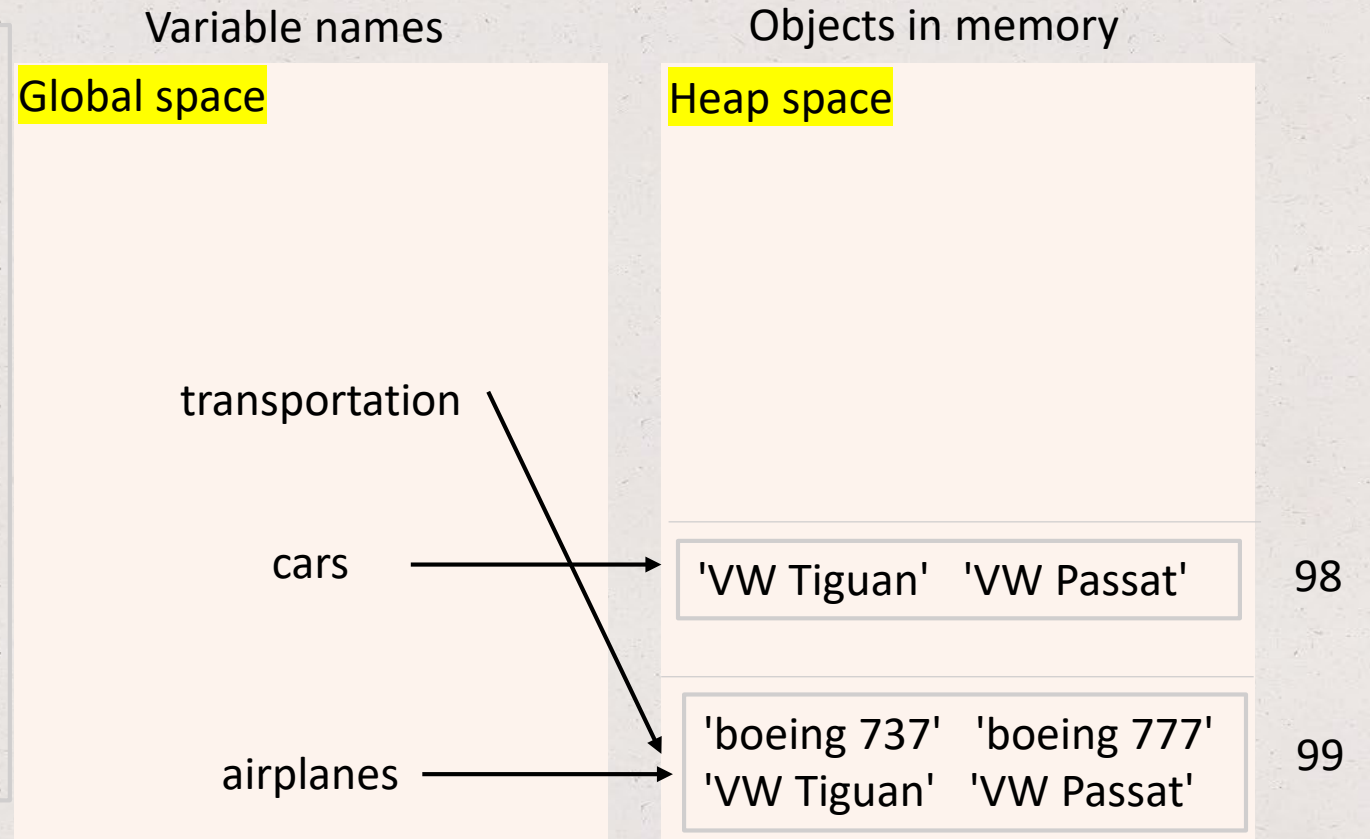


'boeing 737' 'boeing 777'
'VW Tiguan' 'VW Passat'

99

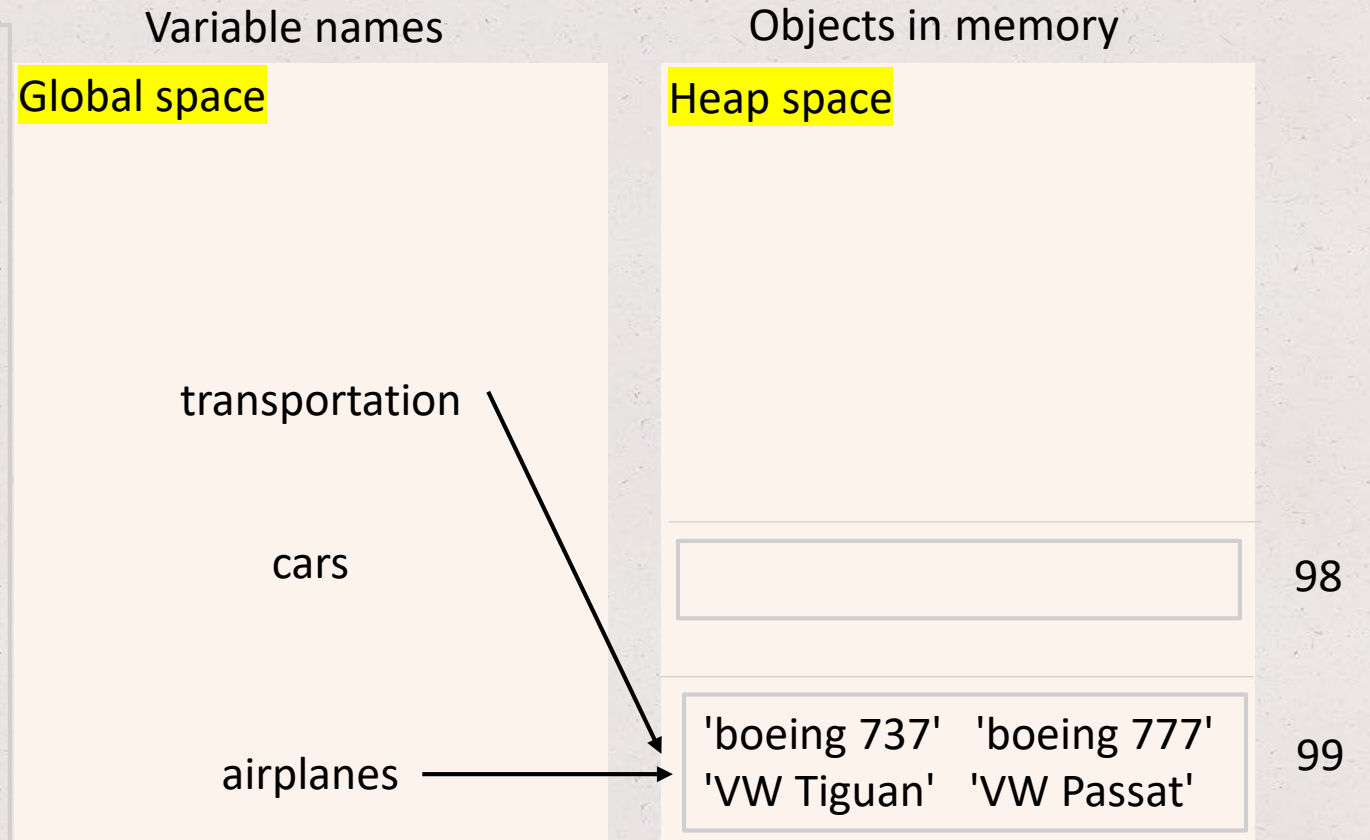
Set basics – memory representation

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



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()
```



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

Global space

transportation

cars

airplanes

Objects in memory

Heap space

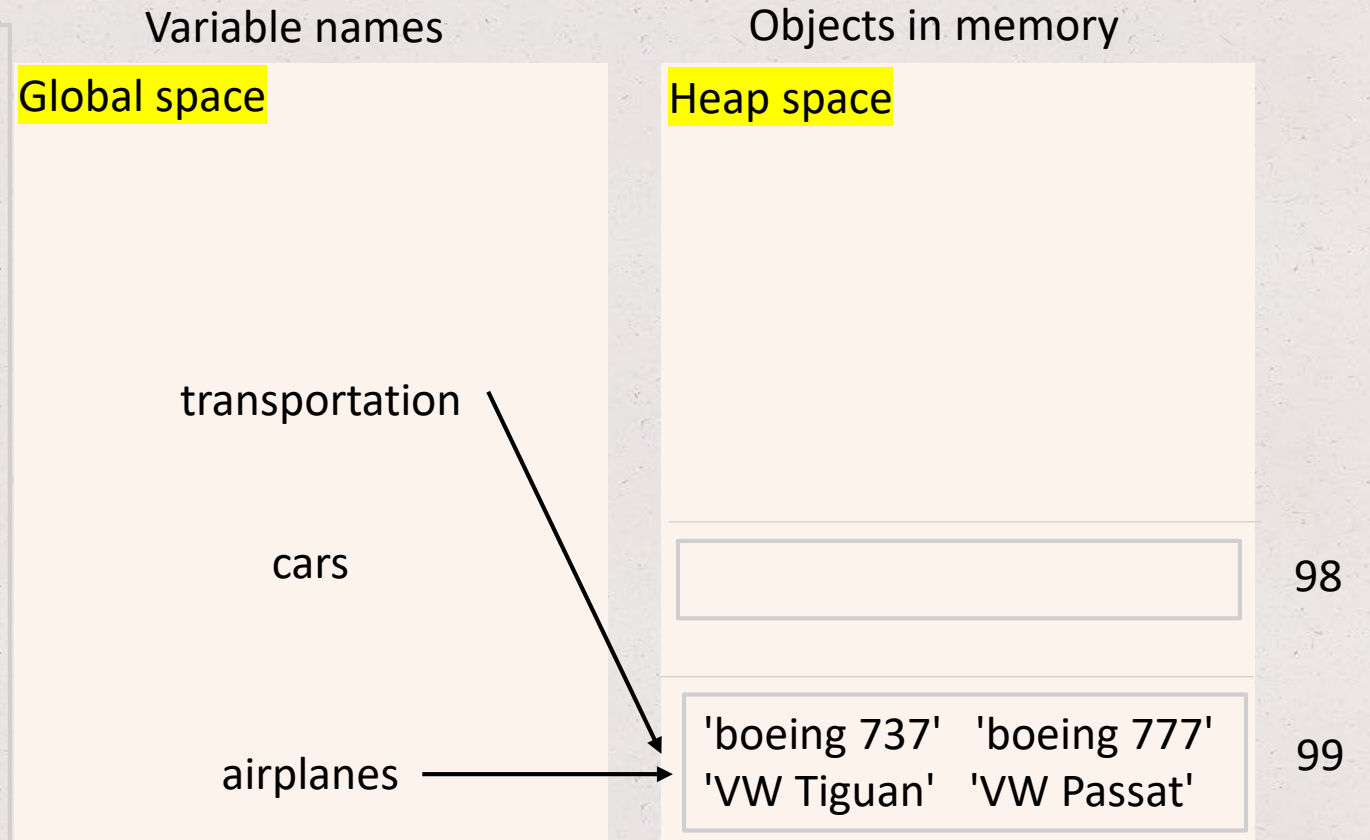
98

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()
```



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

Global space

transportation

cars

airplanes

Objects in memory

Heap space

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>

Top Hat Question # 23

Add question related to sets

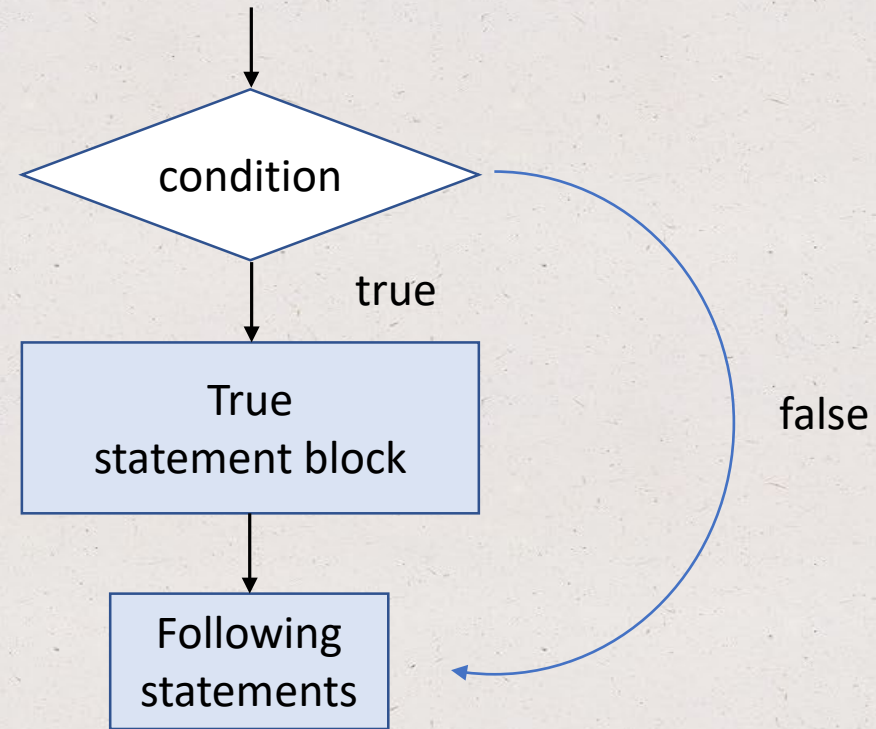
Chapter 4: Branches

- Conditional statements
- Boolean statements
- Membership and identity operators
- Code blocks and indentation
- Conditional expressions

Conditional statements

- The **if** statement:

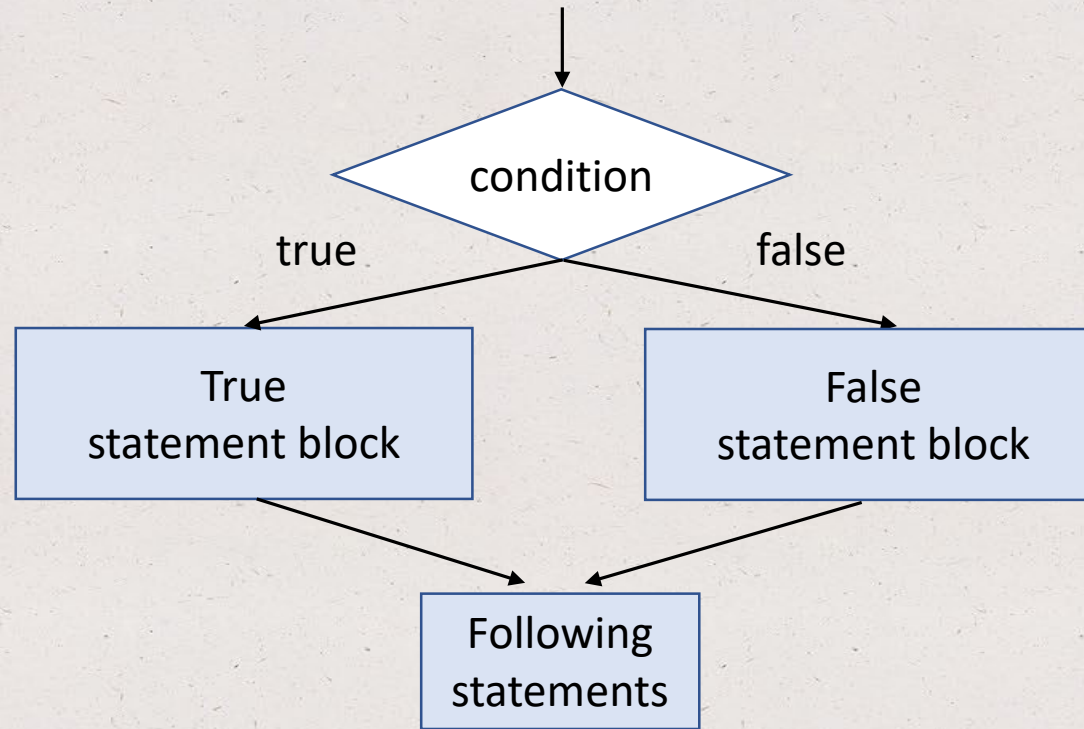
```
if (condition):  
    true statement  
    .  
    .  
    last true statement
```



Conditional statements

- The **if-else** statement:

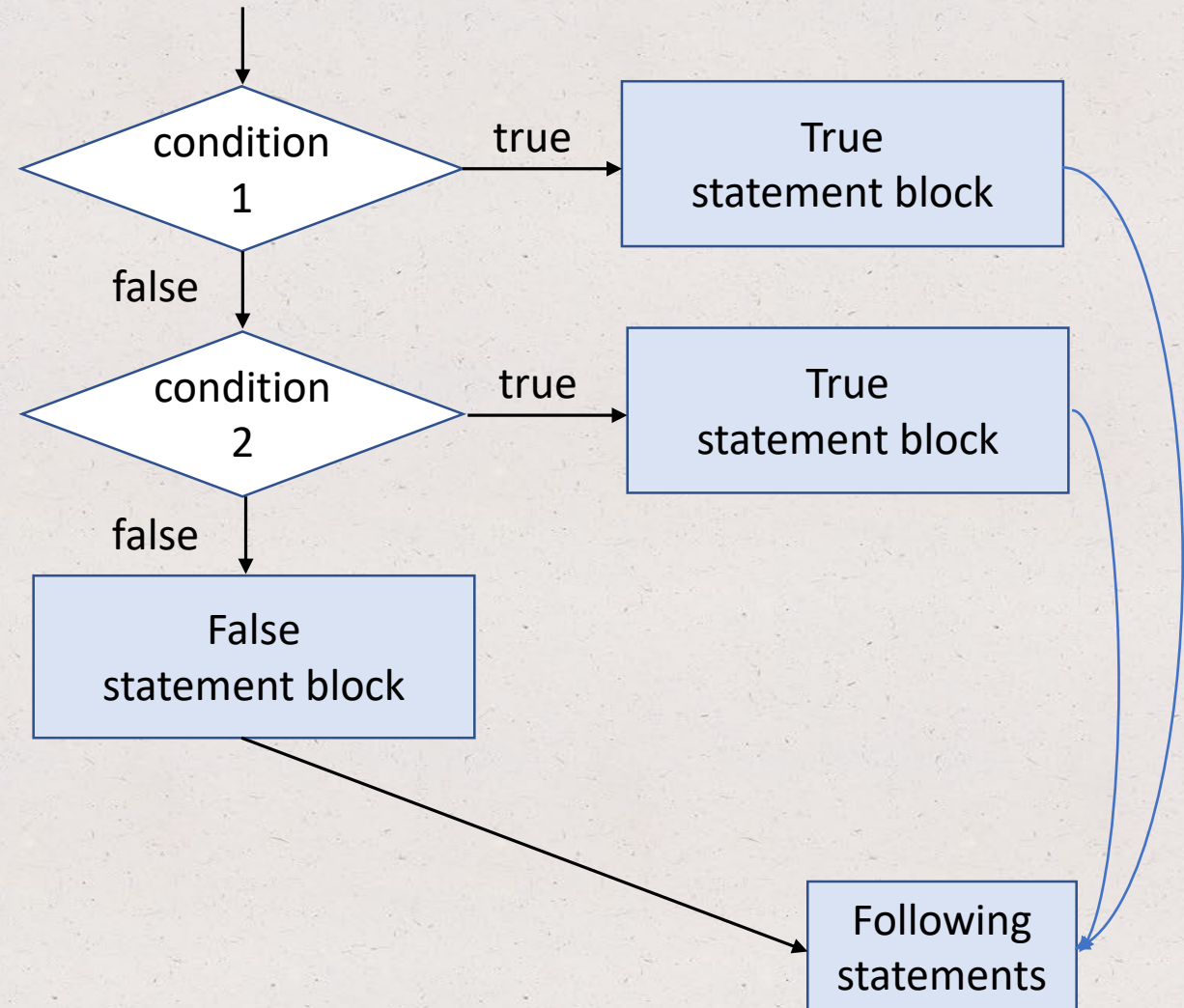
```
if (condition):  
    true statement  
    .  
    .  
    last true statement  
else:  
    false statement  
    .  
    .  
    false statement
```



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

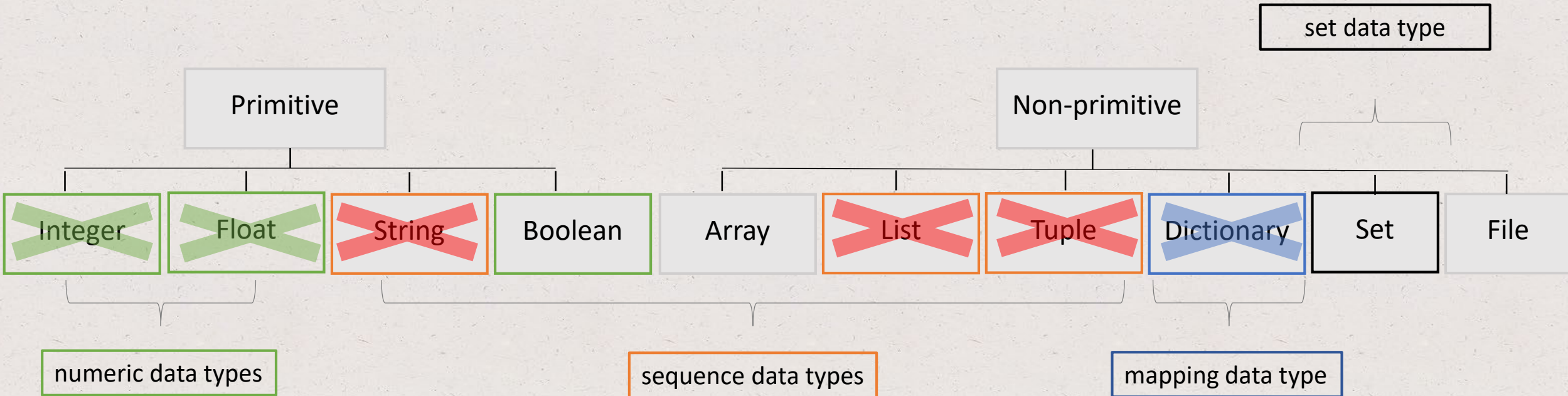
- Boolean values: *True* or *False* (capitalized!)
- Boolean operators: *and*, *or*, *not*
 - **and**: true when both operands are true
 - **or**: true when at least one operand is true
 - **not**: True (False) when the single operand is False (True)
- Boolean expression: an expression that uses Boolean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

More on 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

- **Operator chaining:** You can rent a car if you are between 21 and 81 years old (inclusive)

```
age = int(input('What is your age?: '))  
if (age >= 21) and (age <= 81):  
    print('You can rent a car')
```

25

Boolean operator is **and**

Chaining performs comparisons left to right:

(age >= 21) is True

(age <= 81) is True

The Boolean expression evaluates to **True**

The print statement is: You can rent a car

Top Hat Question # 24

What is printed out in the following code block?

```
to_be = False
if (to_be or (not to_be)):
    print('Tautology')
else:
    print('Contradiction')
```


Boolean statements

- What **data types** can be compared?
 - *Integers and floating-points*
 - *Strings*
 - *Lists and tuples*
 - *Dictionaries*

Boolean statements

- What **data types** can be compared?
 - *Integers and floating-points*: arithmetically compared

```
2 > 3  
2.0 == 2.0
```

```
2 > 3 is False  
floating-number are imprecisely represented. ==, !=  
comparisons are not recommended
```

- Strings: compared by converting each character to a number value (ASCII or Unicode) and then comparing each character in order

```
'Tue' == 'Wed'
```

```
T == W: 84 == 87 is False
```

Source: https://en.wikipedia.org/wiki/List_of_Unicode_characters

Boolean statements

- What **data types** can be compared?
 - *Lists and tuples*: compared via an ordered comparison of every element in the sequence

```
price1 = [50, 60, 75]  
price2 = [50, 60, 85]  
price1 == price2
```

```
price1[1] == price2[1] is True  
price1[2] == price2[2] is True  
price1[3] == price2[3] is False  
The expression evaluates to False
```

- Dictionaries: compared by sorting the keys and values of each dictionary and then comparing them as lists

```
products1 = {'GM seed': 120, 'Conv' : 87}  
products2 = {'GM seed': 120, 'Conv' : 88}
```

See next page for solution

Boolean statements

- What **data types** can be compared?
 - Dictionaries: compared by sorting the keys and values of each dictionary and then comparing them as lists

```
products1 = {'GM seed': 120, 'Conv' : 87}  
products2 = {'GM seed': 120, 'Conv' : 88}
```

Notes:

sorted() is a built-in function
dictionary.keys() is a dictionary method
dictionary.values() is a dictionary method

```
#return dictionary keys  
keys1 = sorted(products1.keys())  
keys2 = sorted(products2.keys())  
  
# return dictionary values  
value1 = sorted(products1.values())  
print(value1)  
value2 = sorted(products2.values())  
print(value2)  
  
# print messages  
if(keys1 == keys2):  
    print("The keys are the same")  
    if(value1 == value2):  
        print('The values are also the same')  
    else:  
        print('The values are not the same')  
else:  
    print('The keys are not the same')
```


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

Source: Zybooks, Table 4.7.1

Top Hat Question # 25

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, '.')
```


Top Hat Question # 26

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')
```

Membership operators

- Membership operators: **in** / **not in**
- Why are membership operators important? Or how we can make use of them?

Membership operators

- Membership operators: **in** / **not in**
- Why are membership operators important? Or how we can make use of them?
- Many times we want to know if a specific value exists in a container (eg. list, tuple, dictionary), or if a substring is part of a string
- For example, what do you do if you only want to keep the rows in a list for which prices > 99?
- **in** / **not in** operands yield True or False if the left operand matches the value on the right side

Membership operators

- Membership operators: **in** / **not in**
- Why are membership operators important? Or how we can make use of them?
- Many times we want to know if a specific value exists in a container (eg. list, tuple, dictionary)
- For example, what do you do if you only want to keep the rows in a list for which prices > 99?
- **in** / **not in** operands yield True or False if the left operand matches the value on the right side

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

```
<every element in the list is checked for the value of 99>
True
```


Top Hat Question # 27

What is the output of the print statement?

```
prices = [100, 55, 66, 75, '99']  
return_value = 99 in prices  
print('It is', return_value)
```

Top Hat Question # 28

Write code to check if 'GM seeds' is in the *products* dictionary keys. Print statement if True is 'We are lucky'. Print statement if False is 'Try next time!'

```
products = {'GM seed': 120, 'Conv' : 87}
```

```
'''
```

```
Your solution here
```

```
'''
```


Top Hat Question # 28

Write code to check if 'GM seeds' is in the *products* dictionary keys. Print statement if True is 'We are lucky'. Print statement if False is 'Try next time!'

```
products = {'GM seed': 120, 'Conv' : 87}
if 'GM seed' in products.keys():
    print('We are lucky')
else:
    print('Try next time!')
```

Identity operators

- Identity operators: **is / is not**
- Why are identity operators important? Or how we can make use of them?

Identity operators

- Identity operators: **is / is not**
- Why are identity operators important? Or how we can make use of them?
- What if we want to determine if two variables are the same object (i.e. share the same value)?
- Identity operators *do not* compare object value! They compare object identities, which is the memory address of the object.
- Identity operators return True if the operands reference the same object

```
prices = [100, 55, 66, 75, 99]
return_value = 99 is prices[4]
print(return_value)
```

```
<the interpreter checks if 99 and prices[4] are identical objects>
True
```

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 # 29

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')
```

Conditional expressions

- Similar to **if-else** statements
- But harder to read in general...

1. expression_when_true **if** condition **else** expression_when_false
salary = 1000 if (age >= 18) else salary = 0

2. The if-else equivalent is:

```
if(age <= 18):  
    salary = 1000  
else:  
    salary = 0
```


References

- **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