Introduction to Python

Ashok Kumar Patel

Course objectives

- To study object oriented paradigm in Python.
- To develop their skill set using Python.
- To familiarize with the functionalities and applications of Python.

Course outcomes

- Understand and use the Object Oriented paradigm in Python
- Use the IO model in Python to read and write disk files.
- Write Python programs using collections, regular expression, classifying and categorizing text.

Keywords

Data Type, operators, control flow, tuples, set, dictionaries,OOPs, Classes, objects, Open Database Connectivity (ODBC), Postgres, MySQL, GUI, tkinter, Client Server programing, HTML, Common Gateway Interface (CGI)

Syllabus

Introduction to Python

A Brief History of Python, Different Versions, Python 2 vs Python 3, Installing Python, Environment Variables, Executing Python from the Command Line, Editing Python Files, Basic Python Syntax, String Values, String Operators, Numeric Data Types Conversions, Simple Input and Output, Language components - Control Flow structures and Syntax – Relational Operators - Logical Operators - Bit Wise Operators, Python for Windows

Types and Operation

Conditions, boolean logic, logical operators, ranges, Control statements: if-else, loops (for, while), Flow control, Functions, Scoping, Exceptions, Input and output, Modules, Collections, Lists, Tuples, Sets, Dictionaries, Modules, Standard Modules, Regular Expressions, Quantifiers, Basic String Operations

Classes and OOP

Principles of Object Orientation, Classes in Python, Creating Classes, Instance Methods, Access Specification, data modeling, persistent storage of objects, inheritance, polymorphism, operator overloading, abstract classes, exception handling, try block

Syllabus

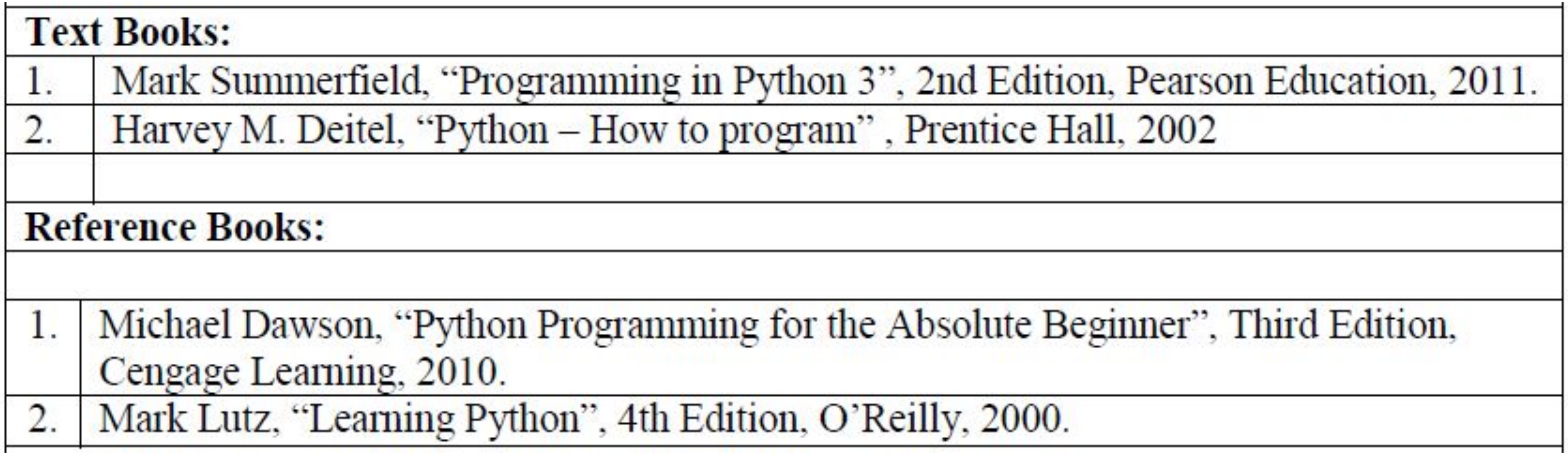
File Handling and Database Programming

File Handling, Writing Data to a File, Reading Data From a File -Additional File Methods: Using Pipes as Data Streams, Handling IO Exceptions, Working with Directories, Metadata, File Organization, Database Programming - Generic Database Connectivity using ODBC, Postgres connection in Python, MySQL connection in Python.

GUI and Internet Programming

Graphical user interfaces, event-driven programming paradigm, tkinter module, creating simple GUI, buttons, labels, entry fields, dialogs, widget attributes - sizes, fonts, colors layouts, nested frames, Multithreading, Networks, and Client/Server Programming, introduction to HTML, interacting with remote HTML server, running html-based queries, downloading pages; CGI programming, programming a simple CGI form.

Books



Text Books:	
1.	Mark Summerfield, “Programming in Python 3”, 2nd Edition, Pearson Education, 2011.
2.	Harvey M. Deitel, “Python – How to program” , Prentice Hall, 2002
Reference Books:	
1.	Michael Dawson, “Python Programming for the Absolute Beginner”, Third Edition, Cengage Learning, 2010.
2.	Mark Lutz, “Learning Python”, 4th Edition, O’Reilly, 2000.

Evaluation

Item / Type of Course	Marks/Weightage
Mid-Term Examination (50 marks)	30
TEE (100 marks)	30
Attendance	5
Class Assessment	-
Assignments	-
Lab Assessment	35
Total Marks	100

TEE Marks (out of 100) ≥ 40
AND
Mid-Term (out of 50) + TEE (Out of 100) ≥ 60 (out of 150)

Attendance % Range	Attendance Marks
Below 75%	0
75 -80%	1
81-85%	2
86-90%	3
91-95%	4
96-100%	5

Minimum % Attendance required: **75%** for Mid Term/TEE

From the date of registration into the course to one day before the start of Mid Term/Last Instructional Day

Evaluation

Continuous Assessment (20 Marks)	Challenging Task (15Marks)
• One assessment from each experiment • 20 marks per experiments • Average marks to be taken	• The final assessment for the lab component • Last Session of the semester

Evaluation

Letter Grade	Grade Point	Remarks	
S	10	Pass in the Course	**Performance Grades**
A	9	Pass in the Course	
B	8	Pass in the Course	
C	7	Pass in the Course	
D	6	Pass in the Course	
E	5	Pass in the Course	
F	0	Failed in the course by not securing the minimum total marks required (TEE Marks < 40% OR (Mid-Term+ TEE < 40%))	
N	0	Absent for TEE / Malpractice in Examinations / Acts of indiscipline/ Debarred from writing TEE	
P	-	Passed in a 'Pass-Fail' course	

Introduction to Python

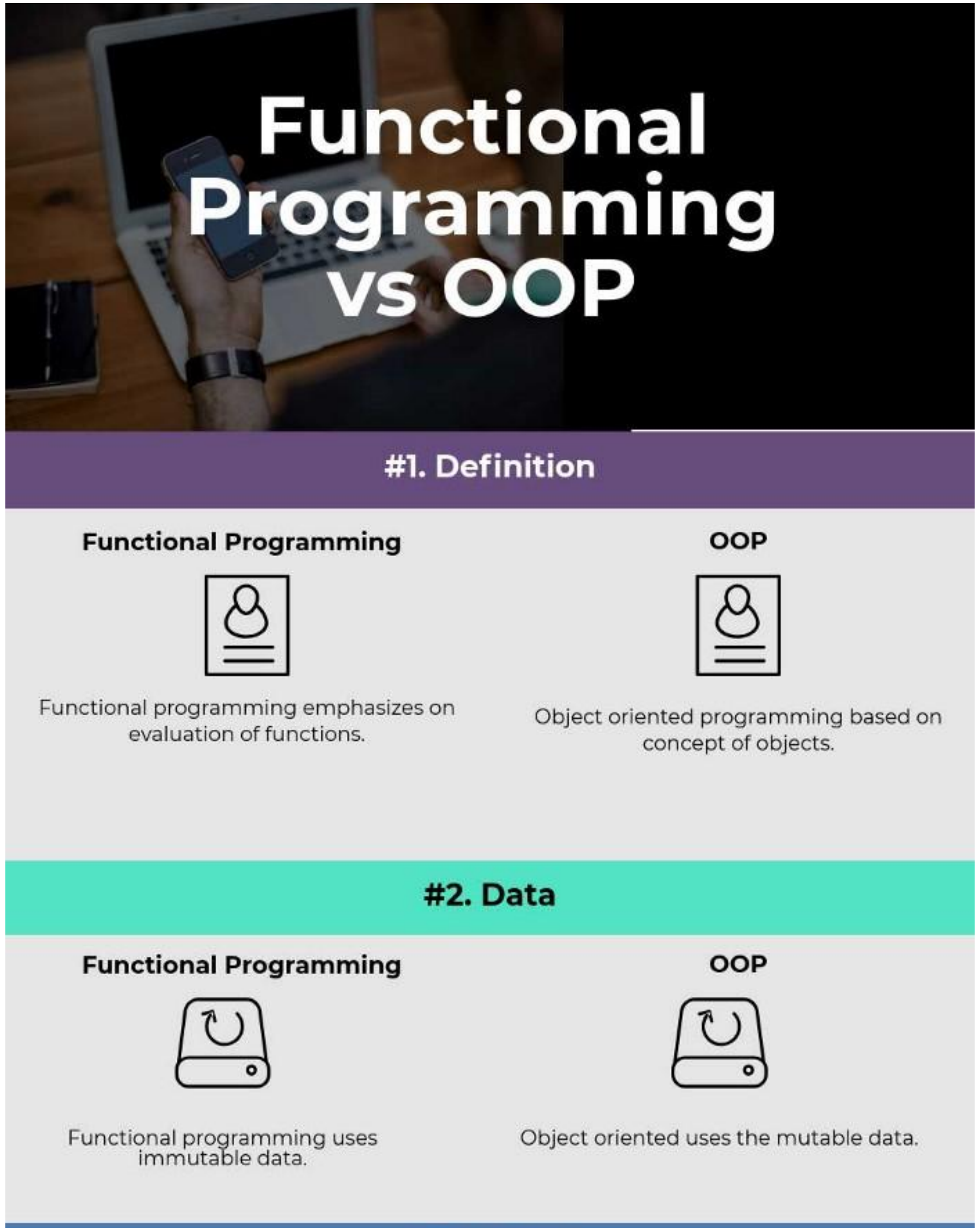
Ashok Kumar Patel

Introduction to Python

What is Python?

Python is a Programming language which is Object-oriented, High-level, Interpreted, Multi-Purpose and Extremely user friendly.

Introduction to Python



Functional Programming vs OOP

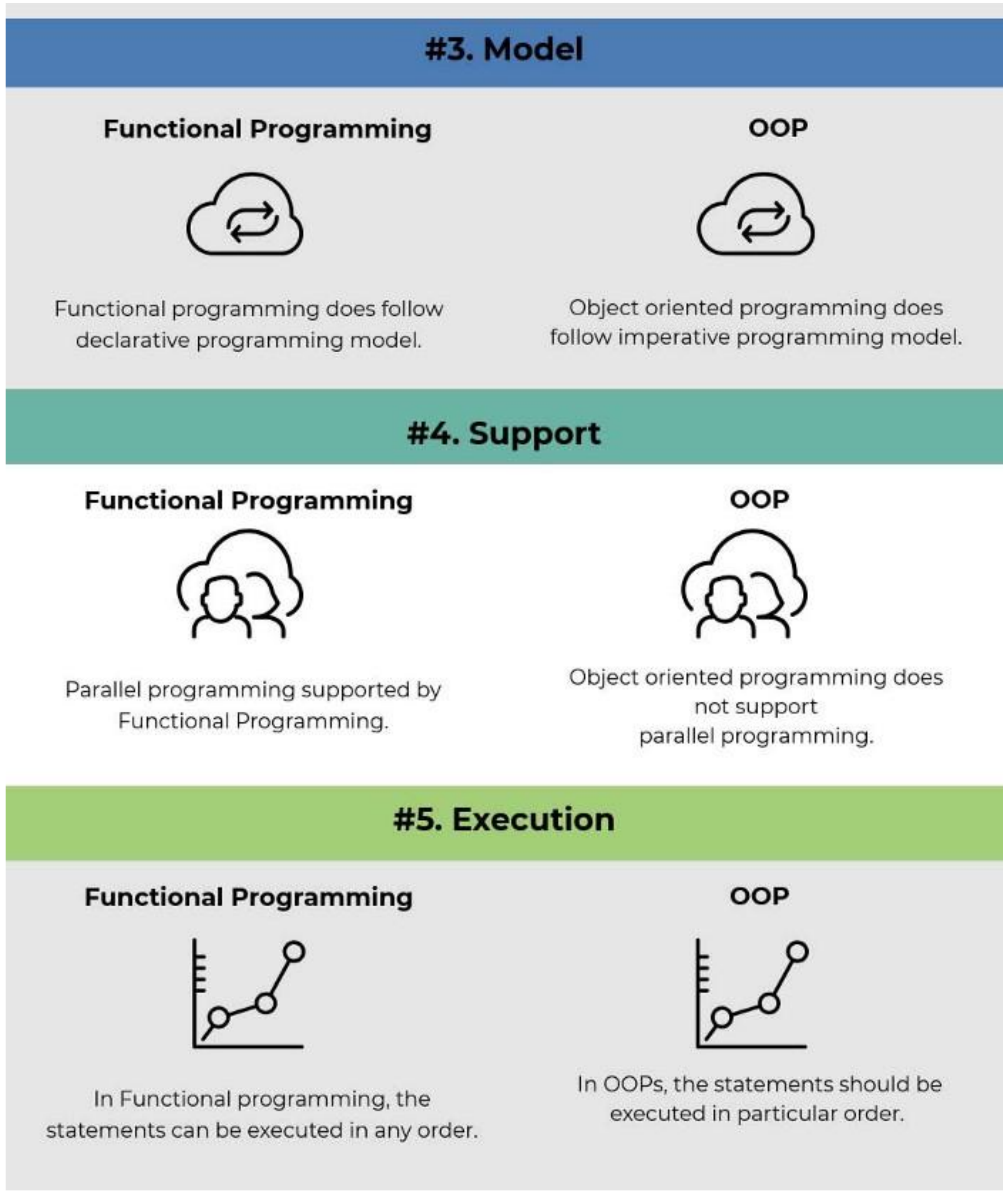
#1. Definition

Functional Programming	OOP
Functional programming emphasizes on evaluation of functions.	Object oriented programming based on concept of objects.

#2. Data

Functional Programming	OOP
Functional programming uses immutable data.	Object oriented uses the mutable data.

Introduction to Python



#3. Model

Functional Programming	OOP
Functional programming does follow declarative programming model.	Object oriented programming does follow imperative programming model.

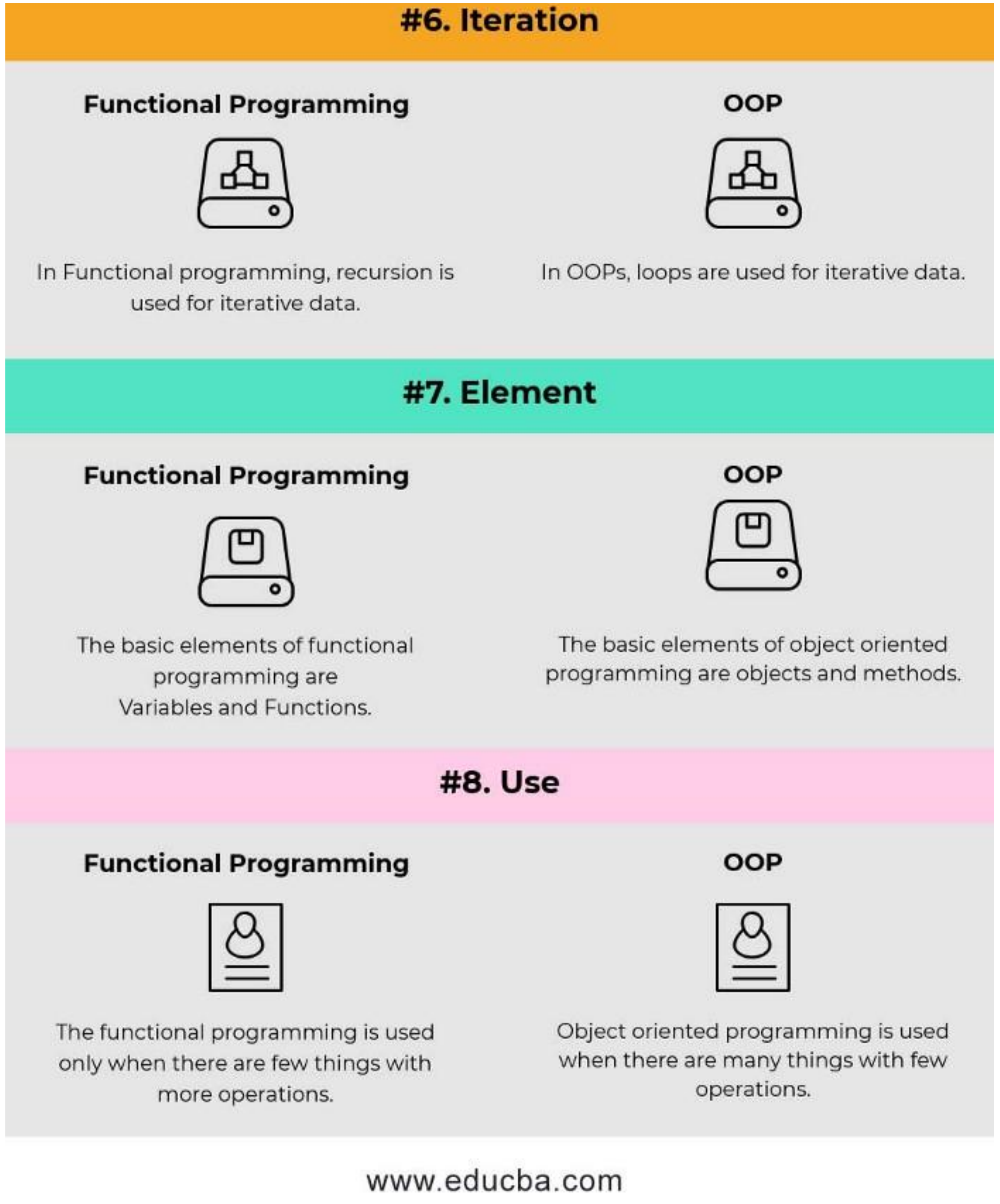
#4. Support

Functional Programming	OOP
Parallel programming supported by Functional Programming.	Object oriented programming does not support parallel programming.

#5. Execution

Functional Programming	OOP
In Functional programming, the statements can be executed in any order.	In OOPs, the statements should be executed in particular order.

Introduction to Python



#6. Iteration

Functional Programming	OOP
In Functional programming, recursion is used for iterative data.	In OOPs, loops are used for iterative data.

#7. Element

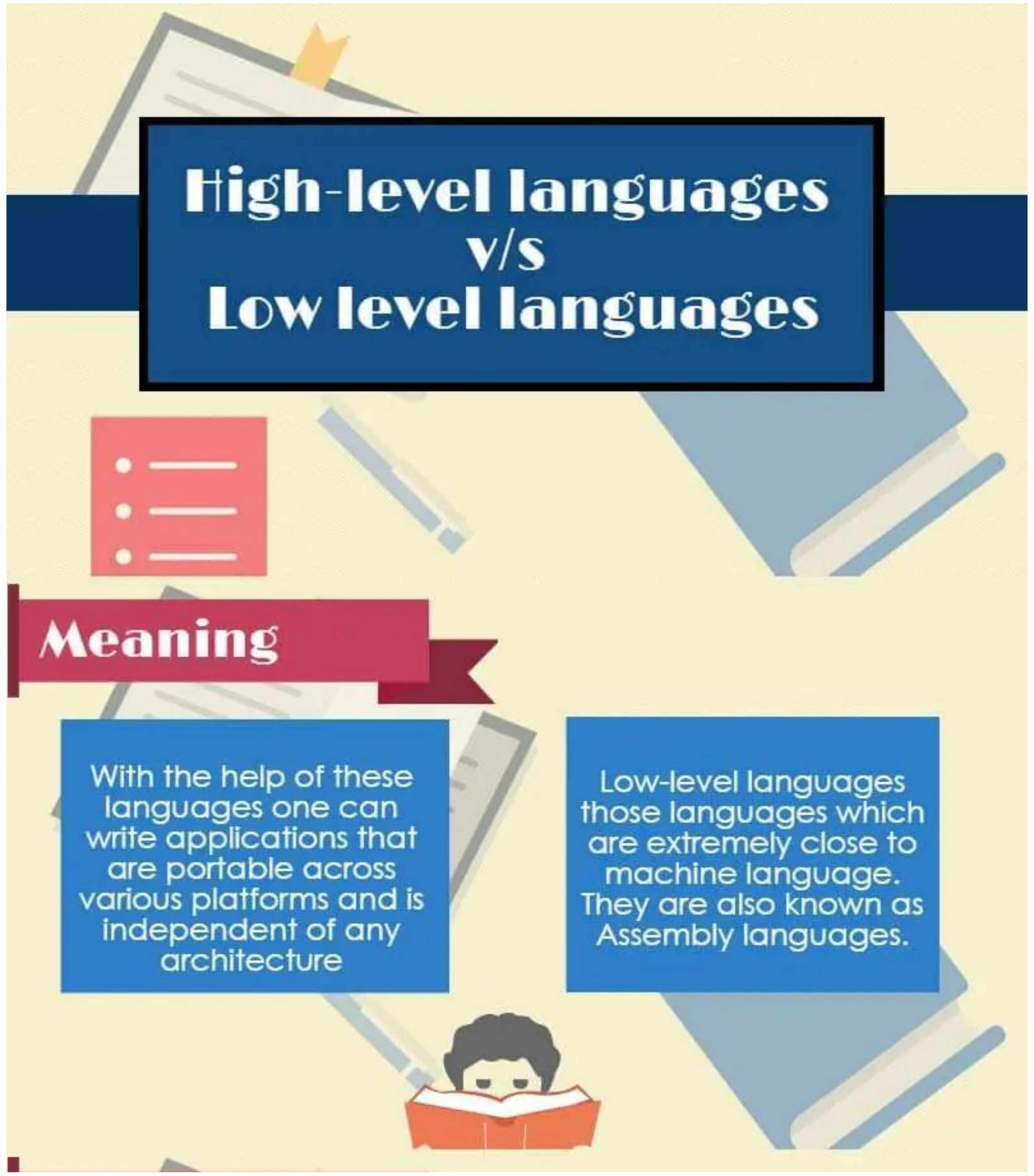
Functional Programming	OOP
The basic elements of functional programming are Variables and Functions.	The basic elements of object oriented programming are objects and methods.

#8. Use

Functional Programming	OOP
The functional programming is used only when there are few things with more operations.	Object oriented programming is used when there are many things with few operations.

www.educba.com

Introduction to Python

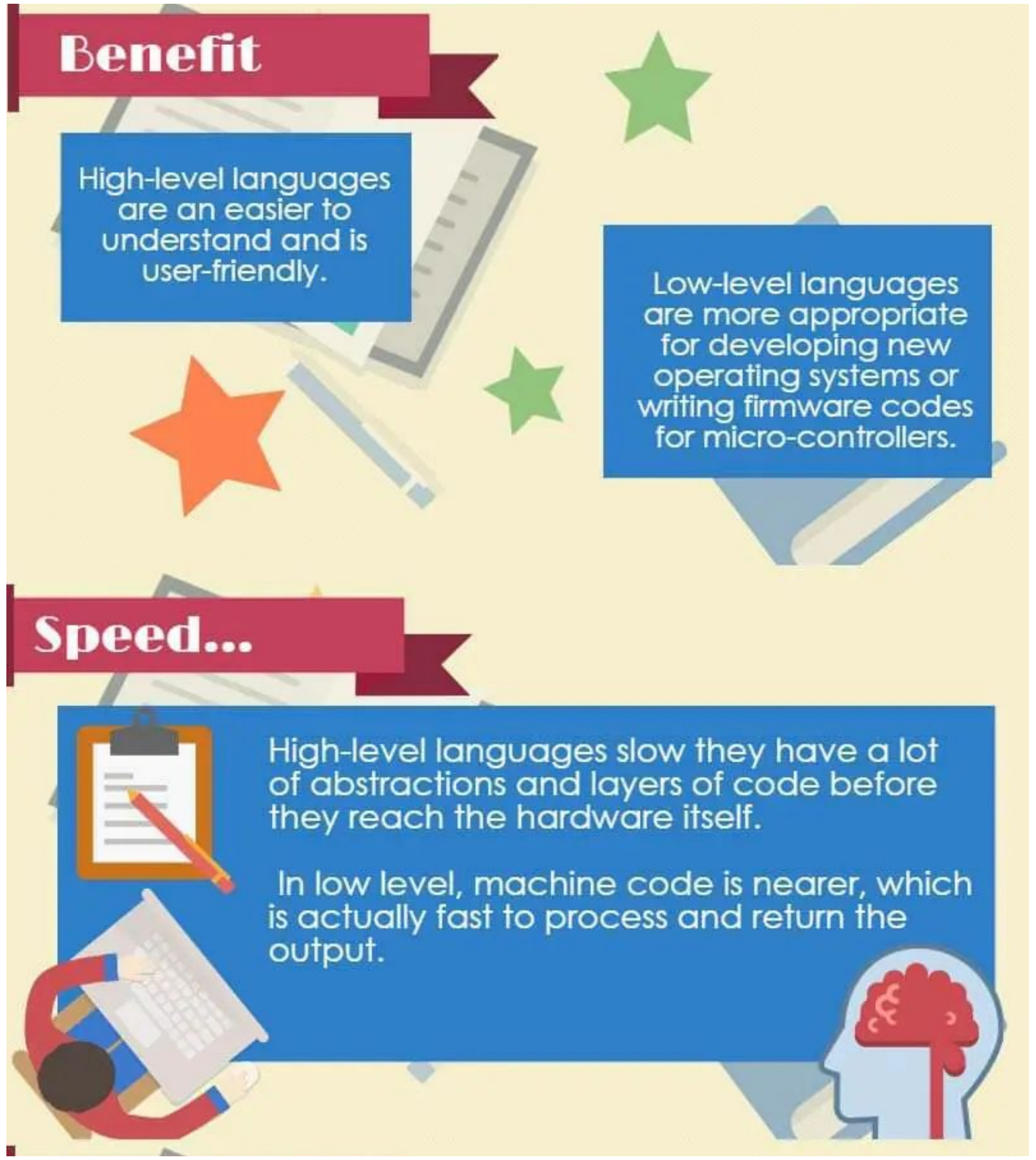

High-level languages
v/s
Low level languages

Meaning

With the help of these languages one can write applications that are portable across various platforms and is independent of any architecture

Low-level languages those languages which are extremely close to machine language. They are also known as Assembly languages.

Introduction to Python



Benefit

High-level languages are an easier to understand and is user-friendly.

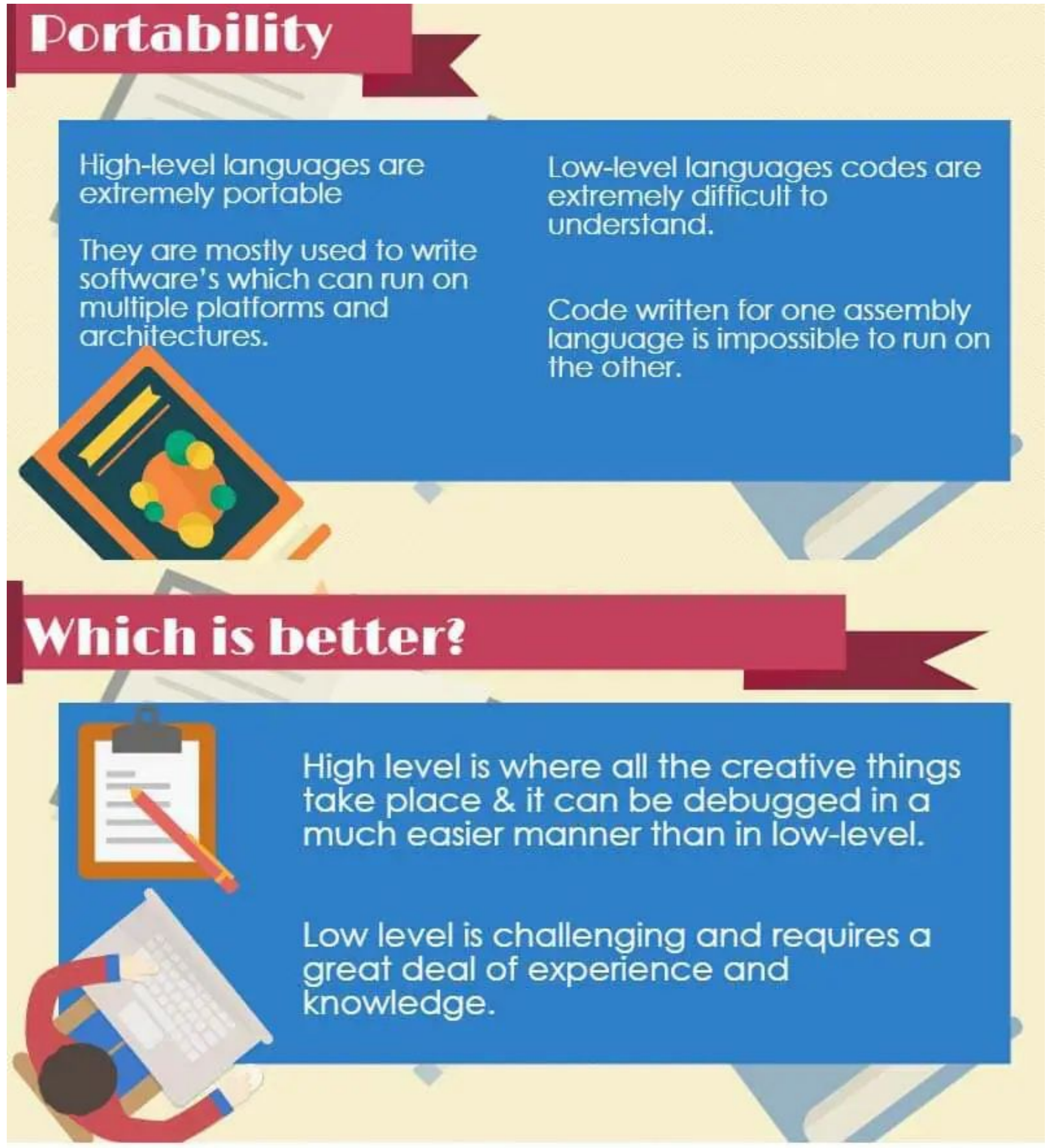
Low-level languages are more appropriate for developing new operating systems or writing firmware codes for micro-controllers.

Speed...

High-level languages slow they have a lot of abstractions and layers of code before they reach the hardware itself.

In low level, machine code is nearer, which is actually fast to process and return the output.

Introduction to Python



Portability

High-level languages are extremely portable

They are mostly used to write software's which can run on multiple platforms and architectures.

Low-level languages codes are extremely difficult to understand.

Code written for one assembly language is impossible to run on the other.

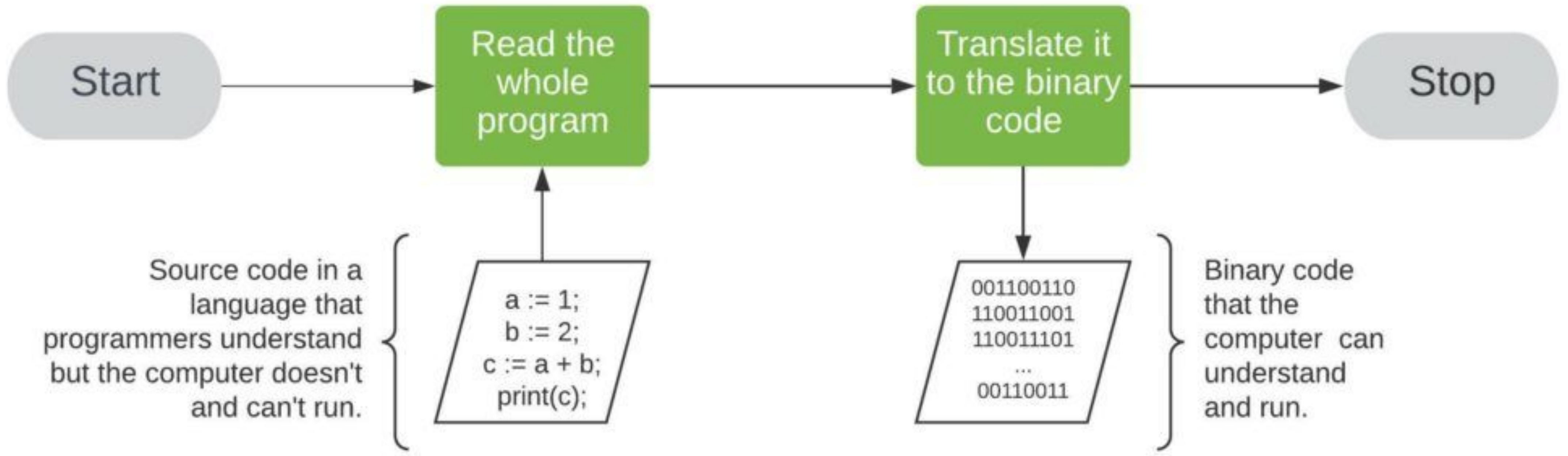
Which is better?

High level is where all the creative things take place & it can be debugged in a much easier manner than in low-level.

Low level is challenging and requires a great deal of experience and knowledge.

Introduction to Python

Computers cannot run the code written in a high–level language. **We first have to translate it into binary code**. To do so, we use compilers and interpreters.

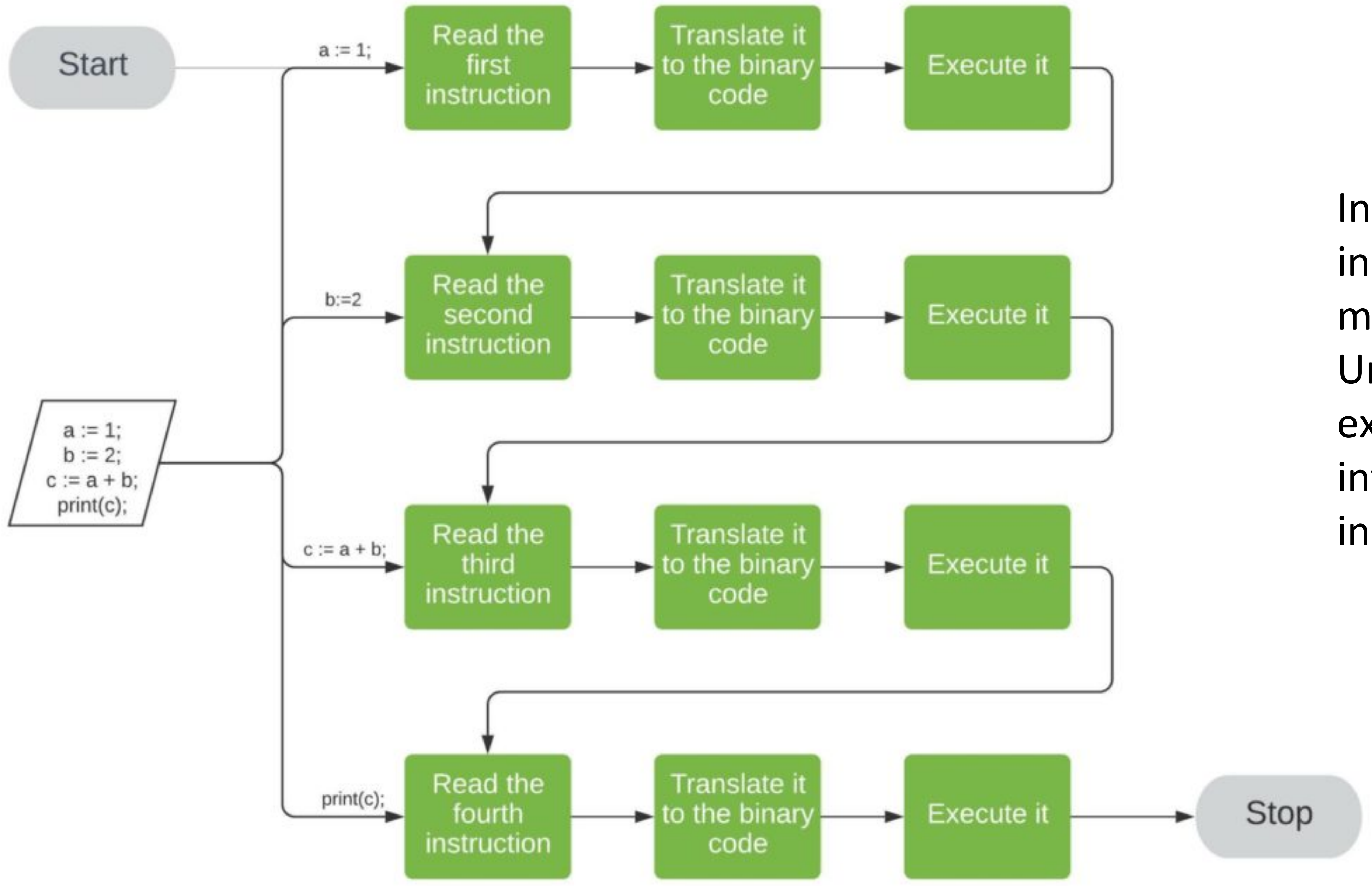

Start
Read the whole program
Translate it to the binary code
Stop
Source code in a language that programmers understand but the computer doesn't and can't run.
a := 1;
b := 2;
c := a + b;
print(c);
001100110
110011001
110011101
...
00110011
Binary code that the computer can understand and run.

Compilers take a whole program as input and translate it to an executable binary code in several steps. We can run the binary code only on the machine on which we compiled it. That's because the binary code depends on the hardware and is not portable.

https://www.baeldung.com/cs/compiled-vs-interpreted-languages

Introduction to Python

Computers cannot run the code written in a high–level language. **We first have to translate it into binary code**. To do so, we use compilers and interpreters.


Start
a := 1;
b := 2;
c := a + b;
print(c);
a := 1;
b:=2
c := a + b;
print(c);
Read the first instruction
Read the second instruction
Read the third instruction
Read the fourth instruction
Translate it to the binary code
Execute it
Stop

Interpreters read and execute the program at hand instruction by instruction. After being read, each instruction is translated into the machine's binary code and run.
Unlike compilers, the interpreters do not produce a binary executable file. Each time we run a program, we invoke the interpreter. It then reads and executes the program one instruction at a time

https://www.baeldung.com/cs/compiled-vs-interpreted-languages

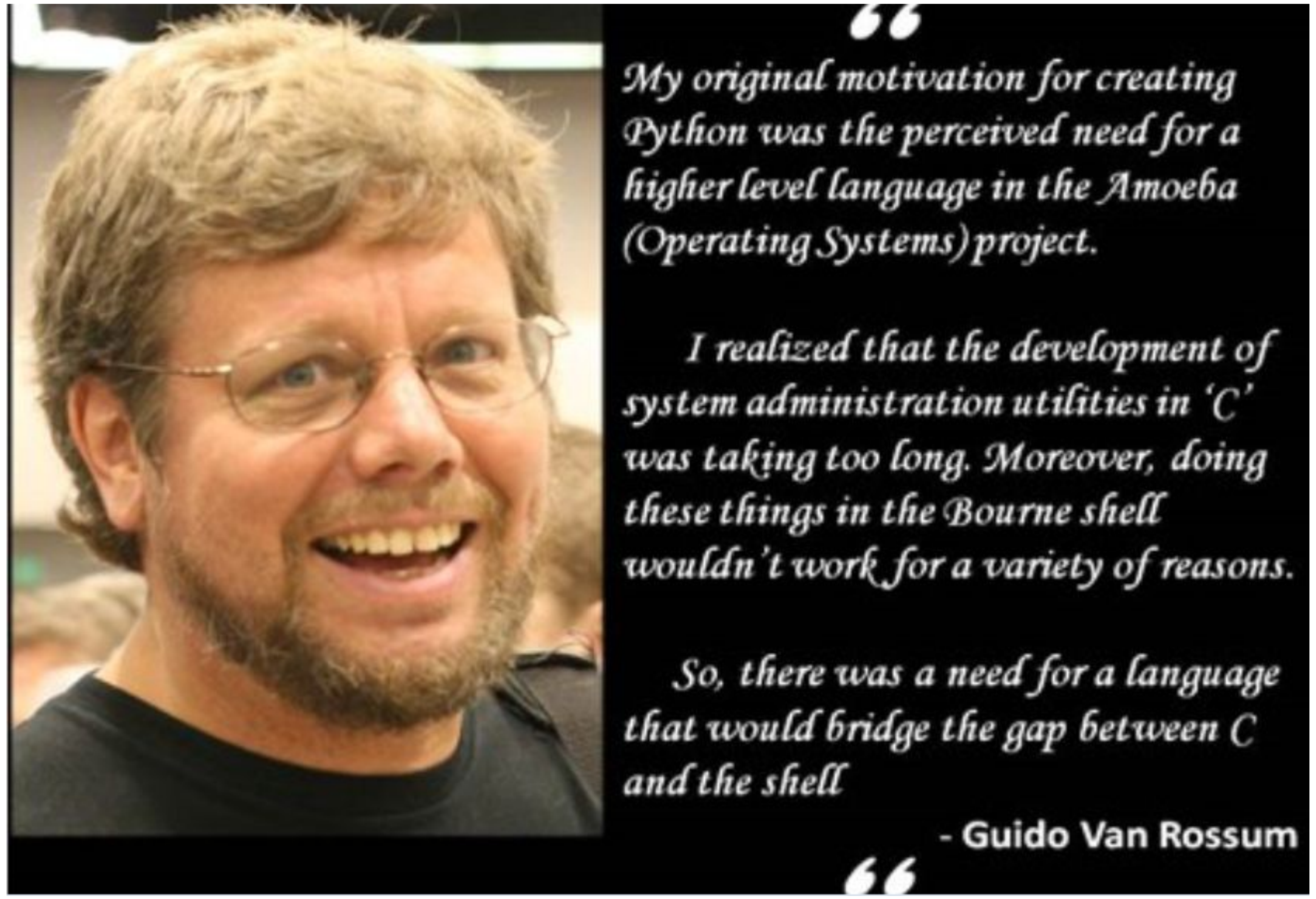
Introduction to Python

History in Brief:

- Invented in the Netherlands by Guido Van Rossum
- Python was coined and conceived in the late 1980s
- Implementation was started in December 1989
- Named after Monty Python
- Guido Van Rossum(the founder) was a fan of Monty Python’s ‘Flying Circus’ – a famous TV show in the Netherlands then.

Introduction to Python

History in Brief:



"My original motivation for creating Python was the perceived need for a higher level language in the Amoeba (Operating Systems) project.

I realized that the development of system administration utilities in 'C' was taking too long. Moreover, doing these things in the Bourne shell wouldn't work for a variety of reasons.

So, there was a need for a language that would bridge the gap between C and the shell"

- Guido Van Rossum

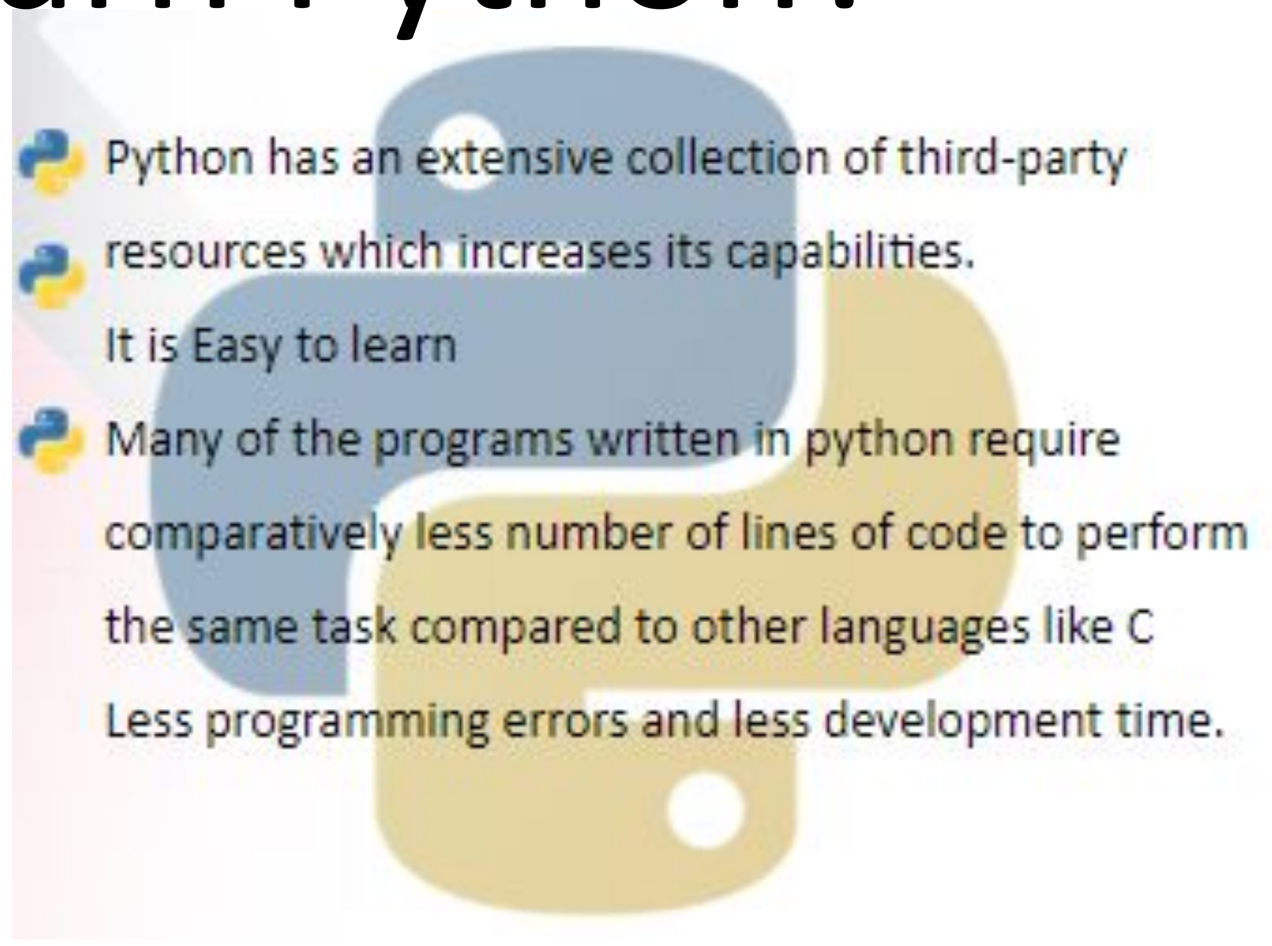
Introduction to Python

Why learn Python?

- Python has a simple syntax; it has abundant libraries and built-in-modules when compared to many other high-level languages like C, C++, Java etc.
- Portable - Python is a cross-platform language i.e. the code written in one operating system, say windows, will work well with other operating systems like Mac OS or Linux.
- It is Easy to use
- Python does a different kind of tasks on varied platforms.

Introduction to Python

Why learn Python?


- Python has an extensive collection of third-party
- resources which increases its capabilities.

  It is Easy to learn
- Many of the programs written in python require

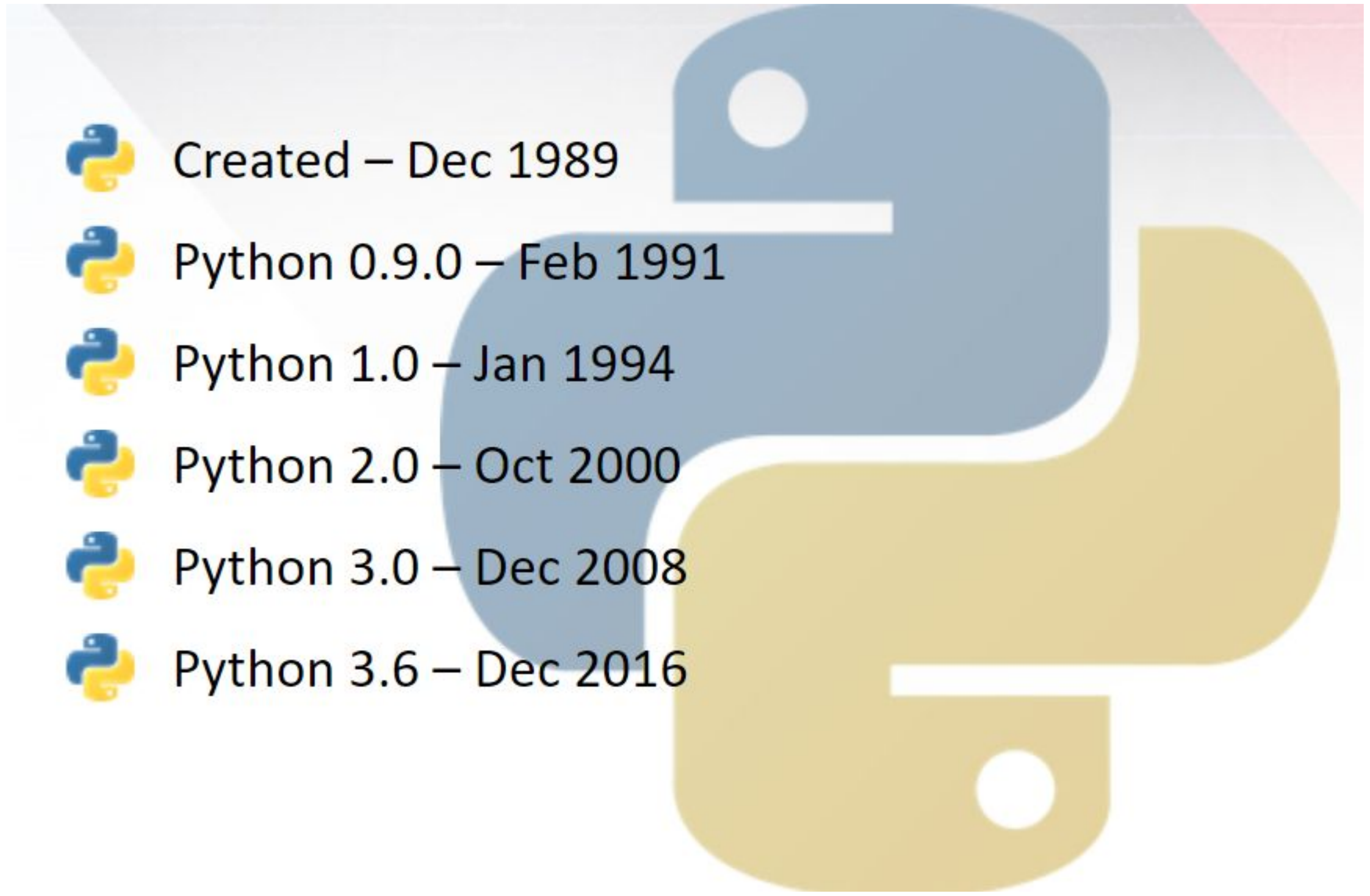
  comparatively less number of lines of code to perform

  the same task compared to other languages like C

  Less programming errors and less development time.

Introduction to Python

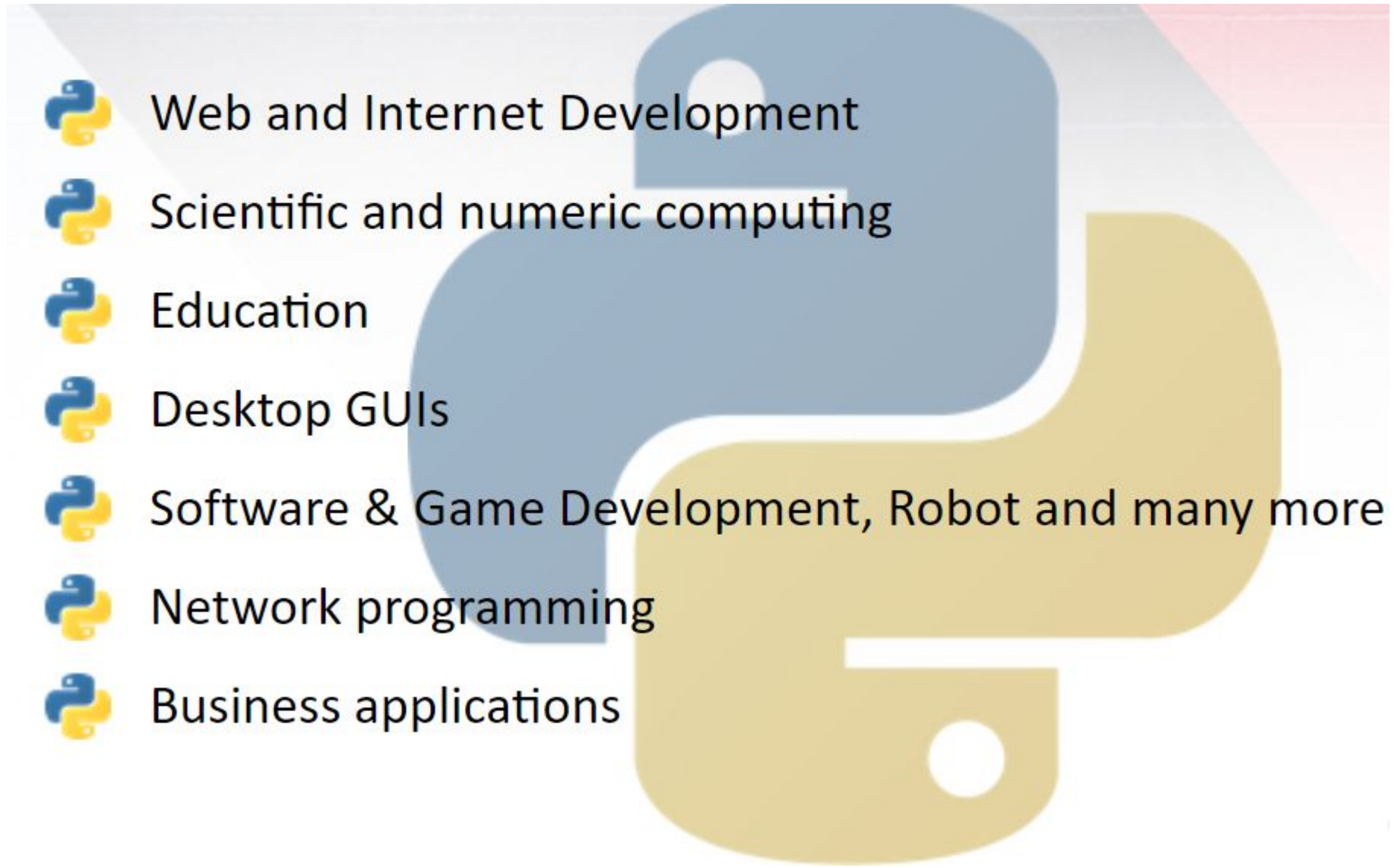
Major Releases:



- Created – Dec 1989
- Python 0.9.0 – Feb 1991
- Python 1.0 – Jan 1994
- Python 2.0 – Oct 2000
- Python 3.0 – Dec 2008
- Python 3.6 – Dec 2016

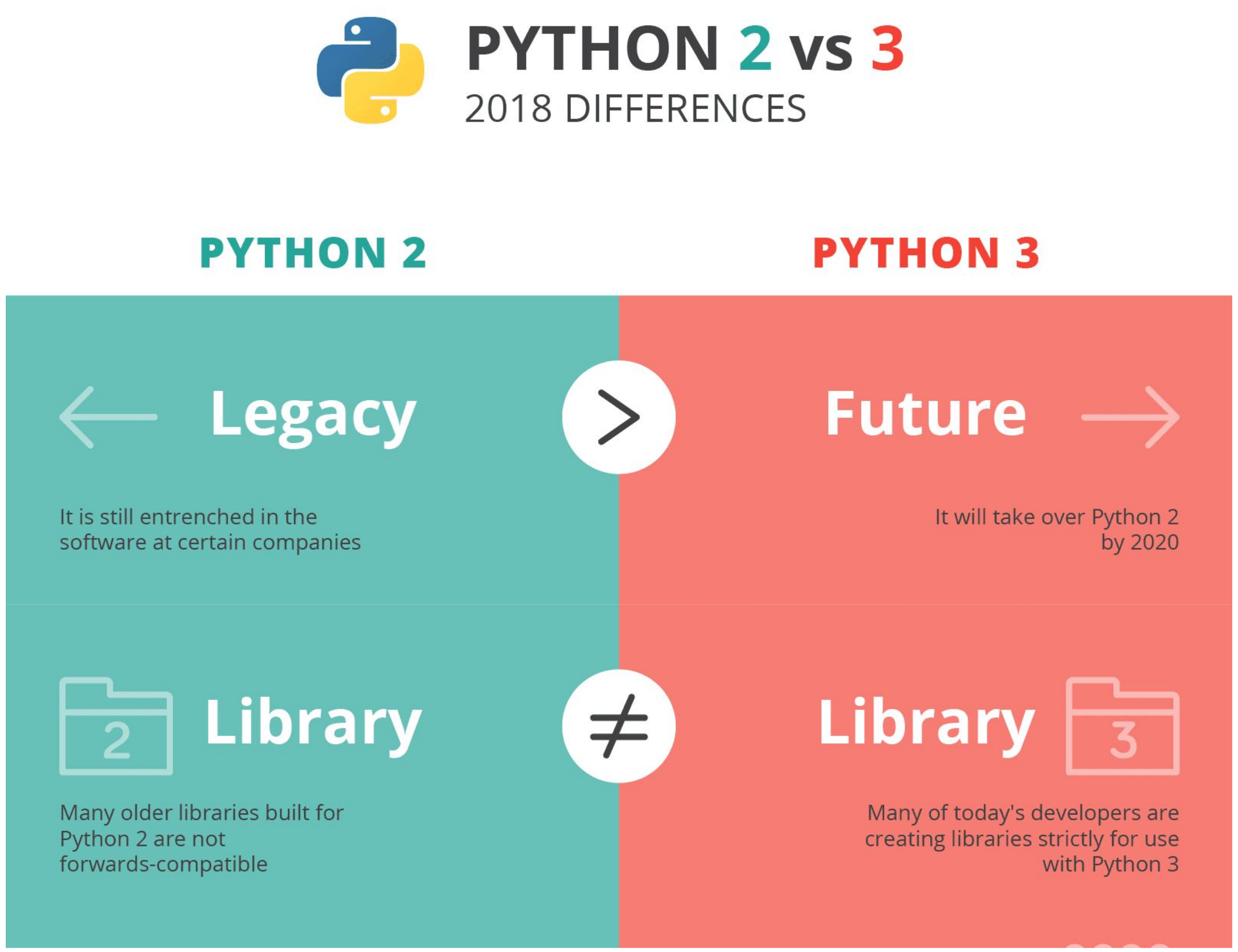
Introduction to Python

Applications:



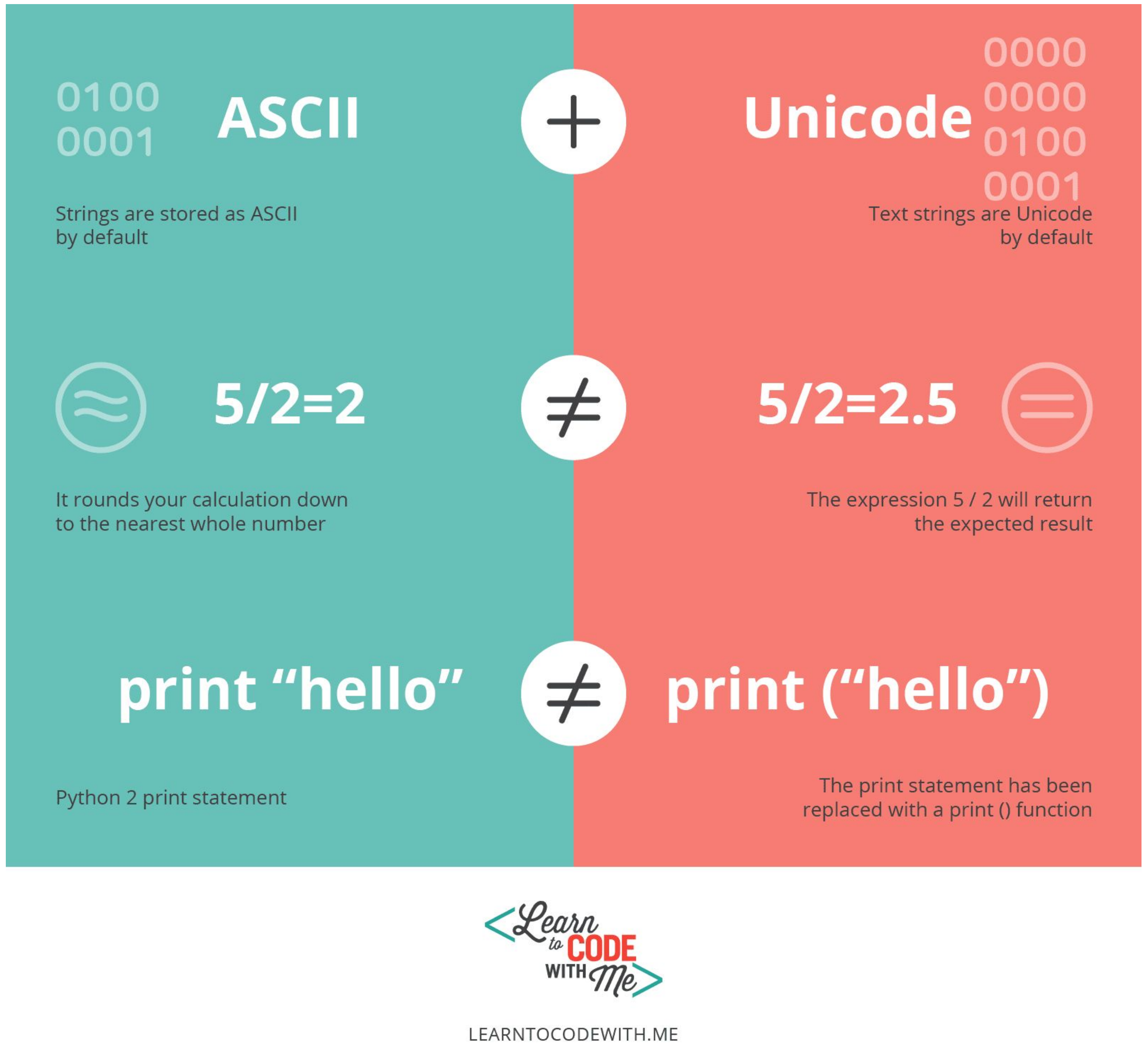
- Web and Internet Development
- Scientific and numeric computing
- Education
- Desktop GUIs
- Software & Game Development, Robot and many more
- Network programming
- Business applications

Introduction to Python



PYTHON 2 vs 3
2018 DIFFERENCES

PYTHON 2		PYTHON 3
Legacy It is still entrenched in the software at certain companies	>	Future It will take over Python 2 by 2020
Library 2 Many older libraries built for Python 2 are not forwards-compatible	≠	Library 3 Many of today's developers are creating libraries strictly for use with Python 3



0100 0001 ASCII Strings are stored as ASCII by default	+	Unicode 0000 0000 0100 0001 Text strings are Unicode by default
5/2=2 It rounds your calculation down to the nearest whole number	≠	5/2=2.5 The expression 5 / 2 will return the expected result
print "hello" Python 2 print statement	≠	print ("hello") The print statement has been replaced with a print () function

Learn to CODE WITH Me
LEARNTOCODEWITH.ME

https://learntocodewith.me/learn/python-2-vs-python-3/

Introduction to Python

Start up a console, and on Windows enter the following commands (which assume that Python is installed in the default location)—the console's output is shown in lightface; what you type is shown in **bold**:

```
C:\>cd c:\py3eg
C:\py3eg\>c:\python31\python.exe hello.py
```

hello.py

```
#!/usr/bin/env python3
print("Hello", "World!")
```

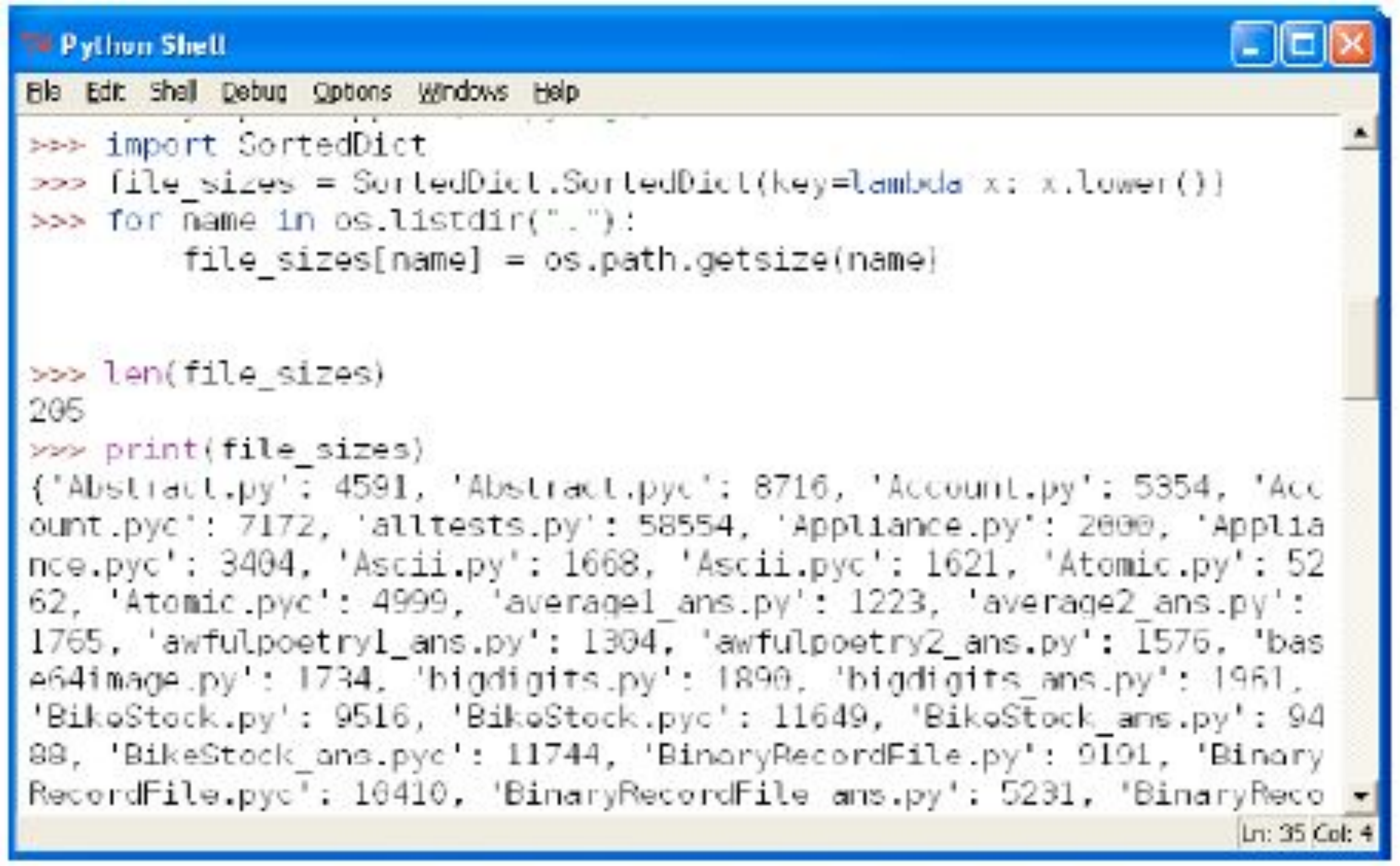
```
$ cd $HOME/py3eg
$ python3 hello.py
```

In both cases the output should be the same:

```
Hello World!
```

Introduction to Python

IDLE provides three key facilities: the ability to enter Python expressions and code and to see the results directly in the Python Shell; a code editor that provides Python-specific color syntax highlighting and indentation support; and a debugger that can be used to step through code to help identify and kill bugs. The Python Shell is especially useful for trying out simple algorithms, snippets of code, and regular expressions, and can also be used as a very powerful and flexible calculator.



```
>>> import SortedDict
>>> file_sizes = SortedDict.SortedDict(key=lambda x: x.lower())
>>> for name in os.listdir("."):
        file_sizes[name] = os.path.getsize(name)


>>> len(file_sizes)
205
>>> print(file_sizes)
{'Abstract.py': 4591, 'Abstract.pyc': 8716, 'Account.py': 5354, 'Acc
ount.pyc': 7172, 'alltests.py': 58554, 'Appliance.py': 2000, 'Applia
nce.pyc': 3404, 'Ascii.py': 1668, 'Ascii.pyc': 1621, 'Atomic.py': 52
62, 'Atomic.pyc': 4999, 'average1_ans.py': 1223, 'average2_ans.py':
1765, 'awfulpoetry1_ans.py': 1304, 'awfulpoetry2_ans.py': 1576, 'bas
e64image.py': 1734, 'bigdigits.py': 1890, 'bigdigits_ans.py': 1961,
'BikeStock.py': 9516, 'BikeStock.pyc': 11649, 'BikeStock_ans.py': 94
88, 'BikeStock_ans.pyc': 11744, 'BinaryRecordFile.py': 9191, 'Binary
RecordFile.pyc': 10410, 'BinaryRecordFile ans.py': 5231, 'BinaryReco
```

Figure *IDLE's Python Shell*

hello.py

```
#!/usr/bin/env python3
print("Hello", "World!")
```

Introduction to Python

Piece #1: Data Types

One fundamental thing that any programming language must be able to do is represent items of data. Python provides several built-in data types, but we will concern ourselves with only two of them for now. Python represents:

- integers (positive and negative whole numbers) using the **int** type, and
- strings (sequences of Unicode characters) using the **str** type.

some examples of integer and string literals:

```
>>> -973
>>> 210624583337114373395836055367340864637790190801098222508621955072
>>> 0
>>> "Infinitely Demanding"
>>> 'Simon Critchley'
>>> 'positivelyαβφ ÷©'
>>> ''
```

is 2^{217}—the size of Python's integers is limited only by machine memory, not by a fixed number of bytes.

Traditionally, Python Shells use **>>>** as their prompt, although this can be changed.

Introduction to Python

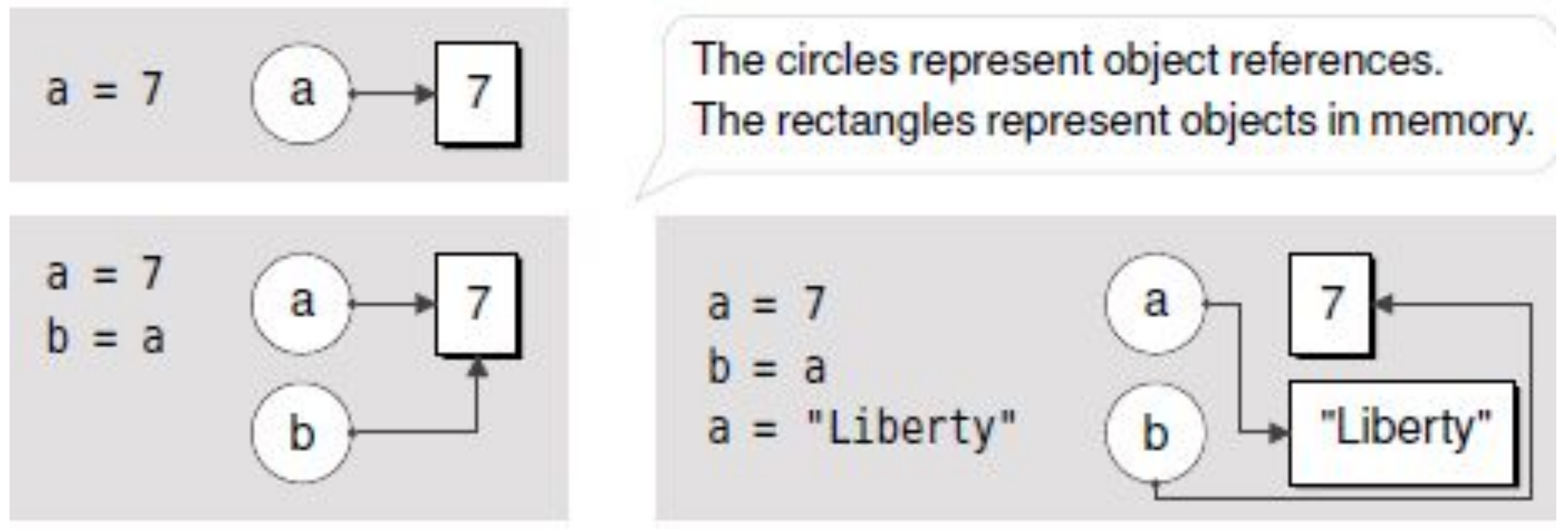
Piece #2: Object References

Once we have some data types, the next thing we need are variables in which to store them. Python doesn't have variables as such, but instead has object references. When it comes to immutable objects like **ints** and **strs**, there is no discernable difference between a variable and an object reference.

some examples :

The syntax is simply *objectReference* = *value*.

```
>>> x = "blue"
>>> y = "green"
>>> z = x
>>> y = x
```



a = 7
a
7
a = 7
b = a
a
7
b
The circles represent object references.
The rectangles represent objects in memory.
a = 7
b = a
a = "Liberty"
a
7
b
"Liberty"

Figure *Object references and objects*

Introduction to Python

Piece #3: Collection Data Types

It is often convenient to hold entire collections of data items. Python provides several collection data types that can hold items, including associative arrays and sets. But here we will introduce just two: **tuple** and **list**.

- Python tuples and lists can be used to hold any number of data items of any data types.
- Tuples are immutable, so once they are created we cannot change them.
- Lists are mutable, so we can easily insert items and remove items whenever we want.

Tuples are created using commas (,), as these examples show

```
>>> "Denmark", "Finland", "Norway", "Sweden"
('Denmark', 'Finland', 'Norway', 'Sweden')
>>> "one",
('one',)
>>> ()
```

Here are some example lists:

```
>>> [1, 4, 9, 16, 25, 36, 49]
>>> ['alpha', 'bravo', 'charlie', 'delta', 'echo']
>>> ['zebra', 49, -879, 'aardvark', 200]
>>> []
```

Introduction to Python

Piece #3: Collection Data Types

Like everything else in Python, collection data types are objects, so we can nest collection data types inside other collection data types, for example, to create lists of lists, without formality.

In procedural programming we call functions and often pass in data items as arguments.

```
>>> len(("one",))
1
>>> len([3, 5, 1, 2, "pause", 5])
6
>>> len("automatically")
13
```

Introduction to Python

Piece #4: Logical Operations

One of the fundamental features of any programming language is its logical operations. Python provides four sets of logical operations, and we will review the fundamentals of all of them here.

The Identity Operator

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

The is operator is a binary operator that returns True if its left-hand object reference is referring to the same object as its right-hand object reference.

```
>>> a = "Something"
>>> b = None
>>> a is not None, b is None
(True, True)
```

The most common use case for is is to compare a data item with the built-in null object, None, which is often used as a place-marking value to signify “unknown” or “nonexistent”:

This process is called "interning". This means that when you write the same value multiple times, it will have the same memory address (id), and you will get the same id for each instance of that value.
A =5, b=5 A is b TRUE

Introduction to Python

Piece #4: Logical Operations

One of the fundamental features of any programming language is its logical operations. Python provides four sets of logical operations, and we will review the fundamentals of all of them here.

Comparison Operators

Python provides the standard set of binary comparison operators, with the expected semantics: < less than, <= less than or equal to, == equal to, != not equal to, >= greater than or equal to, and > greater than.

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

Similarly, strings appear to compare properly too:

```
>>> a = "many paths"
>>> b = "many paths"
>>> a is b
False
>>> a == b
True
```

Introduction to Python

Piece #4: Logical Operations

One of the fundamental features of any programming language is its logical operations. Python provides four sets of logical operations, and we will review the fundamentals of all of them here.

The Membership Operator

```
>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True

>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

Logical Operators

logical operators: and, or, and not

```
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

```
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

not x   Returns True if x is False, False otherwise
x and y Returns x if x is False, y otherwise
x or y Returns y if x is False, x otherwise

Both and and or use short-circuit logic and return the operand that determined the result—they do not return a Boolean (unless they actually have Boolean operands).

Introduction to Python

Piece #5: Control Flow Statements

Instead of progressing line by line, the flow of control can be diverted by a function or method call or by a control structure, such as a conditional branch or a loop statement. Control is also diverted when an exception is raised.

The general syntax for Python’s if statement is this:

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

```
if x:
print("x is nonzero")
```

```
if lines < 1000:
    print("small")
elif lines < 10000:
    print("medium")
else:
    print("large")
```

The general syntax for Python’s while statement is this:

```
while boolean_expression:
    suite
```

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

Introduction to Python

Piece #5: Control Flow Statements

Instead of progressing line by line, the flow of control can be diverted by a function or method call or by a control structure, such as a conditional branch or a loop statement. Control is also diverted when an exception is raised.

The general syntax for Python's for statement is this:

```
for variable in iterable:
    suite
```

```
>>> for country in ["Denmark", "Finland", "Norway", "Sweden"]:
        print(country)
```

```
>>> countries = ["Denmark", "Finland", "Norway", "Sweden"]
>>> for country in countries:
        print(country)
```

The general syntax for Python's try statement (Exception handling) is this:

```
try:
    try_suite
except exception1 as variable1:
    exception_suite1
…
except exceptionN as variableN:
    exception_suiteN
```

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

```
3.5
invalid literal for int() with
base 10: '3.5'

13
valid integer entered: 13
```

Introduction to Python

Piece #6: Arithmetic Operators

Python provides a full set of arithmetic operators, including binary operators for the four basic mathematical operations:+ addition, - subtraction,* multiplication, and / division. In addition, many Python data types can be used with augmented assignment operators such as += and *=.

```
>>> 5 + 6
11
>>> 3 - 7
-4
>>> 4 * 8
32

>>> 12 / 3
4.0
>>> 3 // 2
1
```

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

When an augmented assignment operator is used on an immutable object (int, str) is that the operation is performed, and an object holding the result is created; and then the target object reference is re-bound to refer to the result object rather than the object it referred to before

And if the original object ***a*** was referring to has no more object references referring to it, it will be scheduled for garbage collection.

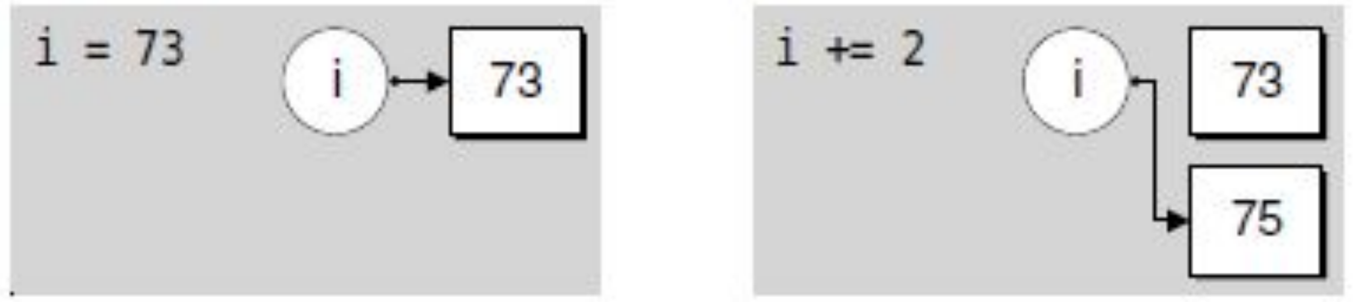


Figure *Augmented assignment of an immutable object*

Introduction to Python

Piece #6: Arithmetic Operators

Python provides a full set of arithmetic operators, including binary operators for the four basic mathematical operations:+ addition, - subtraction,* multiplication, and / division. In addition, many Python data types can be used with augmented assignment operators such as += and *=.

```
>>> seeds = ["sesame", "sunflower"]
>>> seeds += ["pumpkin"]
>>> seeds
['sesame', 'sunflower', 'pumpkin']
```

Since lists are mutable, when += is used the original list object is modified, so no rebinding of seeds is necessary

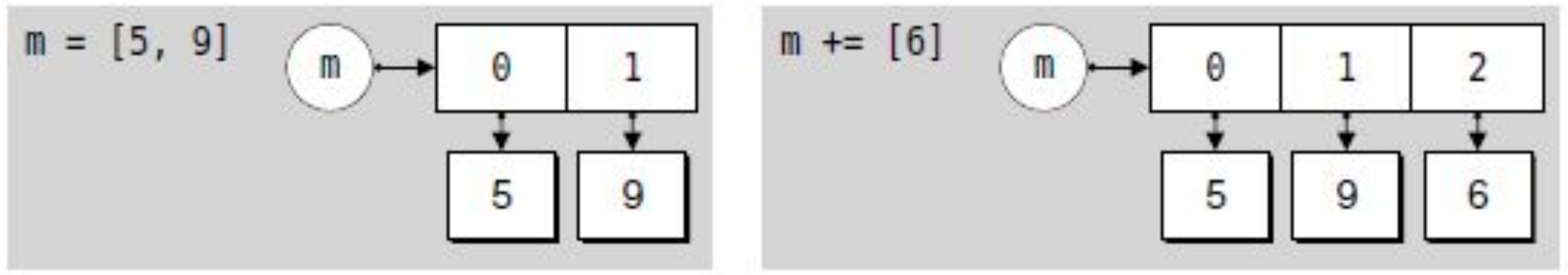


Figure *Augmented assignment of a mutable object*

When an augmented assignment operator is used on an immutable object (int, str) is that the operation is performed, and an object holding the result is created; and then the target object reference is re-bound to refer to the result object rather than the object it referred to before

And if the original object ***a*** was referring to has no more object references referring to it, it will be scheduled for garbage collection.

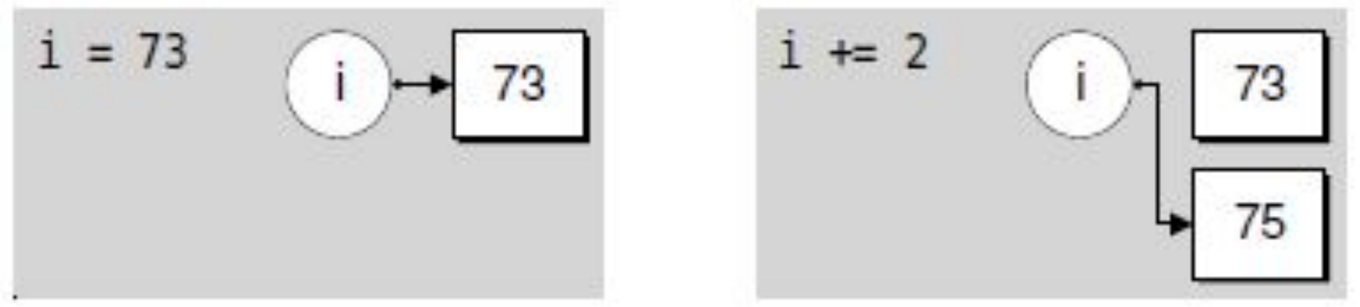


Figure *Augmented assignment of an immutable object*

Introduction to Python

Piece #7: Input/Output

To be able to write genuinely useful programs we must be able to read input—for example, from the user at the console, and from files—and produce output, either to the console or to files.

Python provides the built-in input() function to accept input from the user.

```
print("Type integers, each followed by Enter; or just Enter to finish")
total = 0
count = 0
while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break
if count:
    print("count =", count, "total =", total, "mean =", total / count)
```

```
Type integers, each followed by Enter; or just Enter to finish
number: 12
number: 7
number: 1x
invalid literal for int() with base 10: '1x'
number: 15
number: 5
number:
count = 4 total = 39 mean = 9.75
```

Introduction to Python

Piece #8: Creating and Calling Functions

It is perfectly possible to write programs using the data types and control structures that we have covered in the preceding pieces. However, very often we want to do essentially the same processing repeatedly, but with a small difference, such as a different starting value. Python provides a means of encapsulating suites as functions which can be parameterized by the arguments they are passed.

Here is the general syntax for creating a function:

```
def functionName(arguments):
    suite
```

The *arguments* are optional and multiple arguments must be comma-separated.

Every Python function has a return value; this defaults to None unless we return from the function using the syntax return *value*, in which case *value* is returned. The return value can be just one value or a tuple of values.

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)

age = get_int("enter your age: ")
```

Introduction to Python

Piece #8: Creating and Calling Functions

It is perfectly possible to write programs using the data types and control structures that we have covered in the preceding pieces. However, very often we want to do essentially the same processing repeatedly, but with a small difference, such as a different starting value. Python provides a means of encapsulating suites as functions which can be parameterized by the arguments they are passed.

A Python module is just a .py file that contains Python code, such as custom function and class (custom data type) definitions, and sometimes variables. To access the functionality in a module we must import it.

```
>>> import sys
```

Once a module has been imported, we can access any functions, classes, or variables that it contains.

```
>>> print(sys.argv)
```

```
>>>import random
>>>x = random.randint(1, 6)
>>>y = random.choice(["apple", "banana", "cherry", "durian"])
```

Introduction to Python

Slicing and Striding Strings

Given the assignment `s = "The waxwork man"`, Figure 2.2 shows some example slices for string `s`.

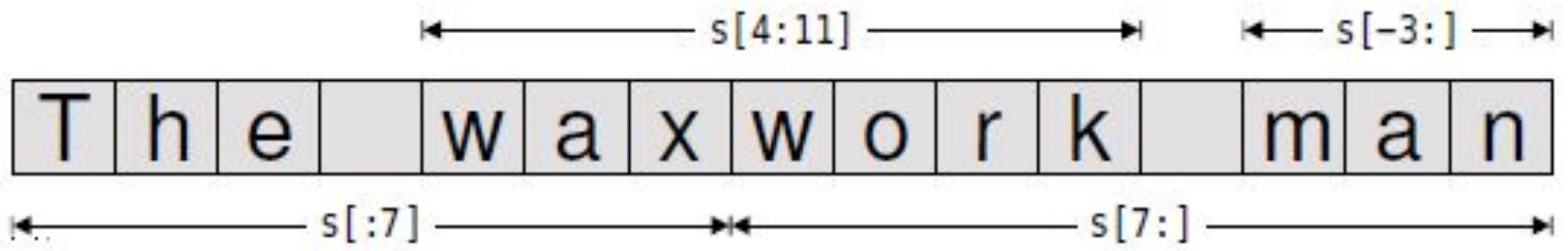
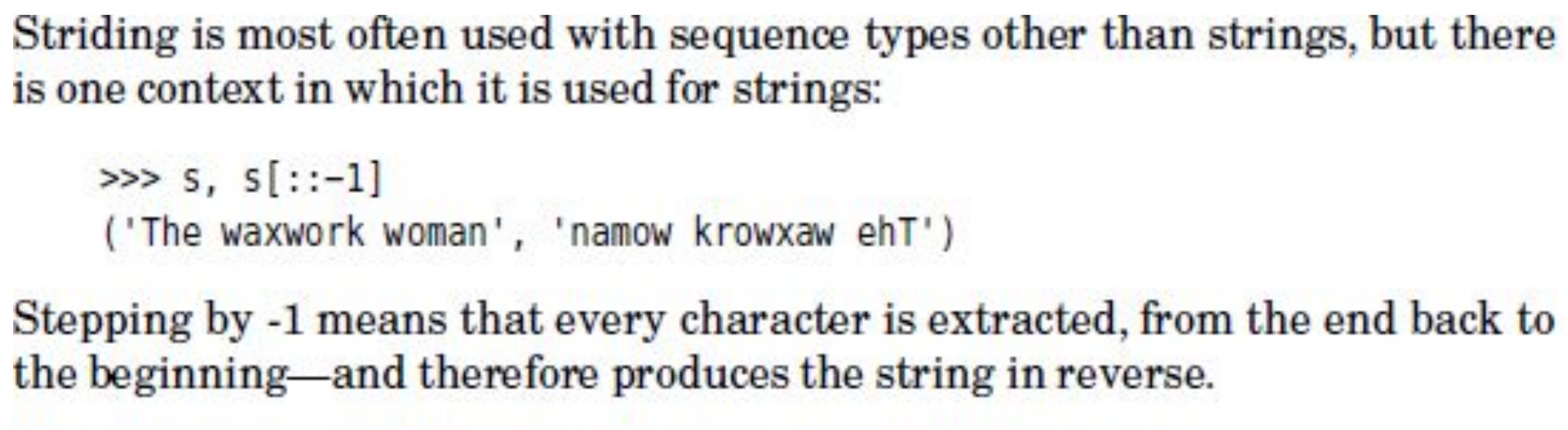


Figure 2.2 *Sequence slicing*

One way of inserting a substring inside a string is to mix slicing with concatenation. For example:

```
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman'
```

Introduction to Python

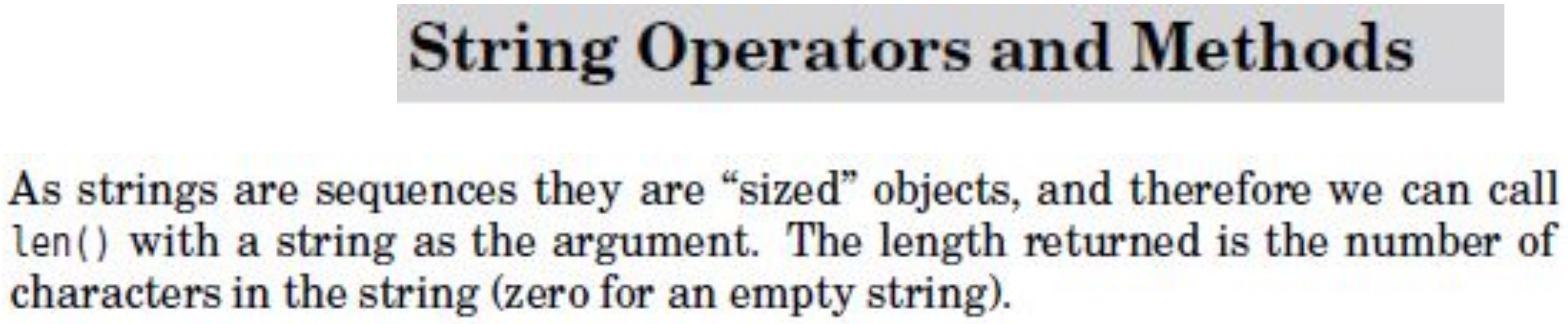


Striding is most often used with sequence types other than strings, but there is one context in which it is used for strings:

```
>>> s, s[::-1]
('The waxwork woman', 'namow krowxaw ehT')
```

Stepping by -1 means that every character is extracted, from the end back to the beginning—and therefore produces the string in reverse.

Introduction to Python



String Operators and Methods

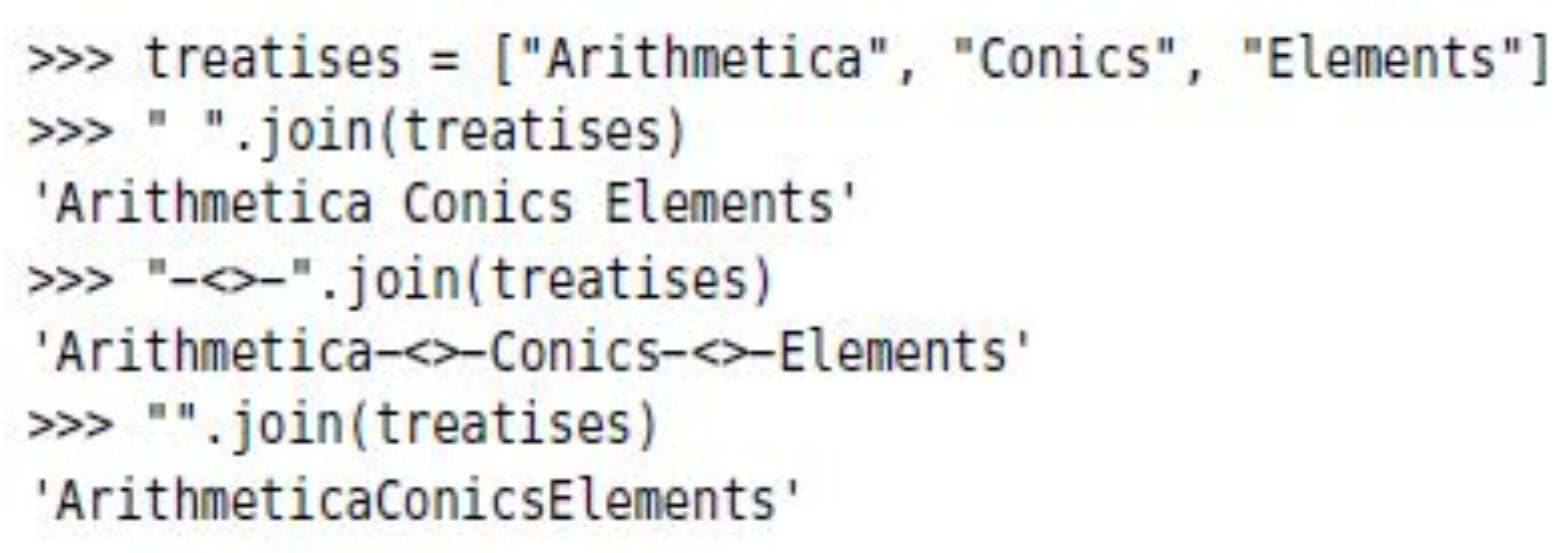
As strings are sequences they are "sized" objects, and therefore we can call `len()` with a string as the argument. The length returned is the number of characters in the string (zero for an empty string).

Concatenation with +,
appending with +=,
replication with *, and
augmented assignment replication with *=.

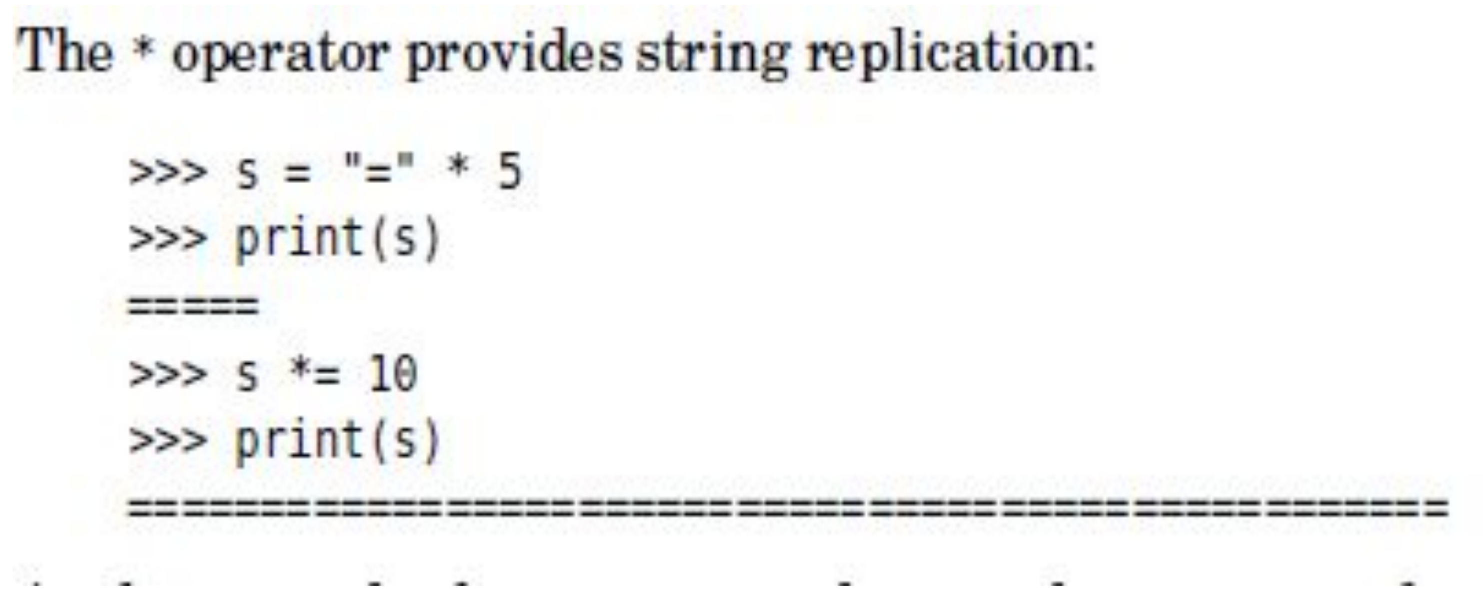
Introduction to Python

String Operators and Methods

In cases where we want to concatenate lots of strings the str.join() method offers a better solution.



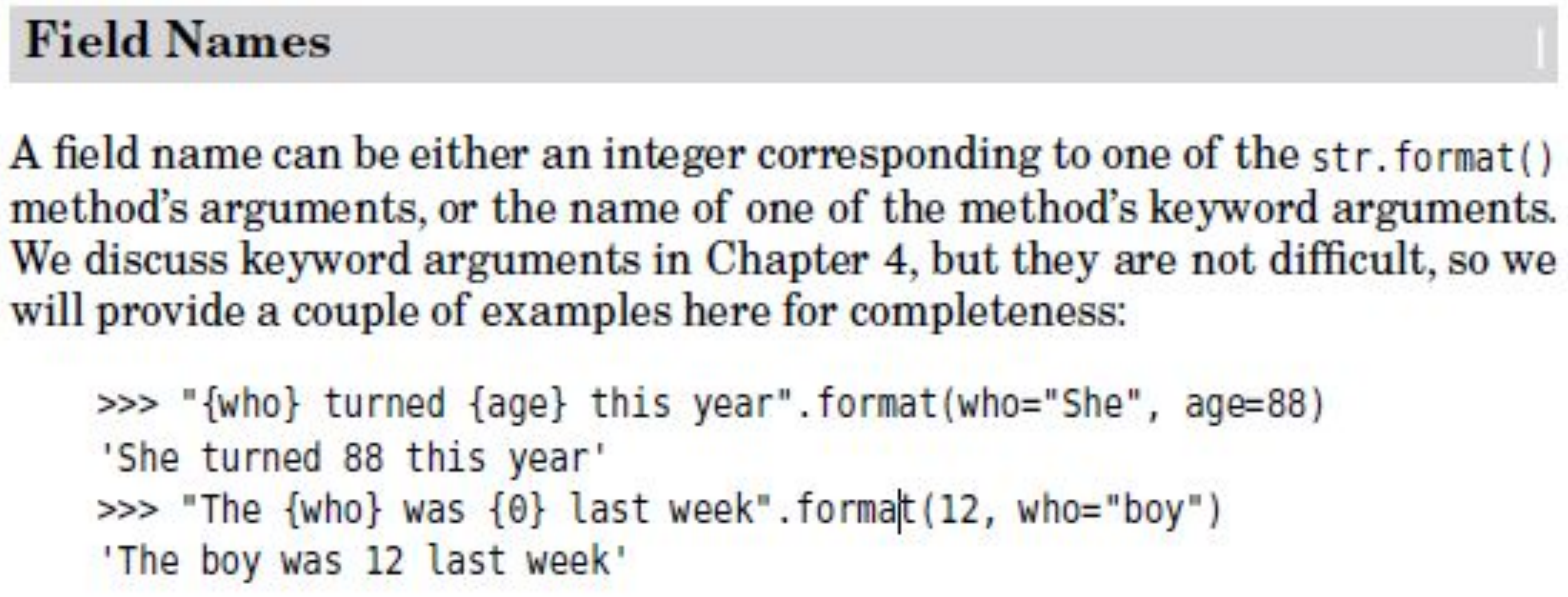
```
>>> treatises = ["Arithmetica", "Conics", "Elements"]
>>> " ".join(treatises)
'Arithmetica Conics Elements'
>>> "-<>-".join(treatises)
'Arithmetica-<>-Conics-<>-Elements'
>>> "".join(treatises)
'ArithmeticaConicsElements'
```



The * operator provides string replication:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

Introduction to Python

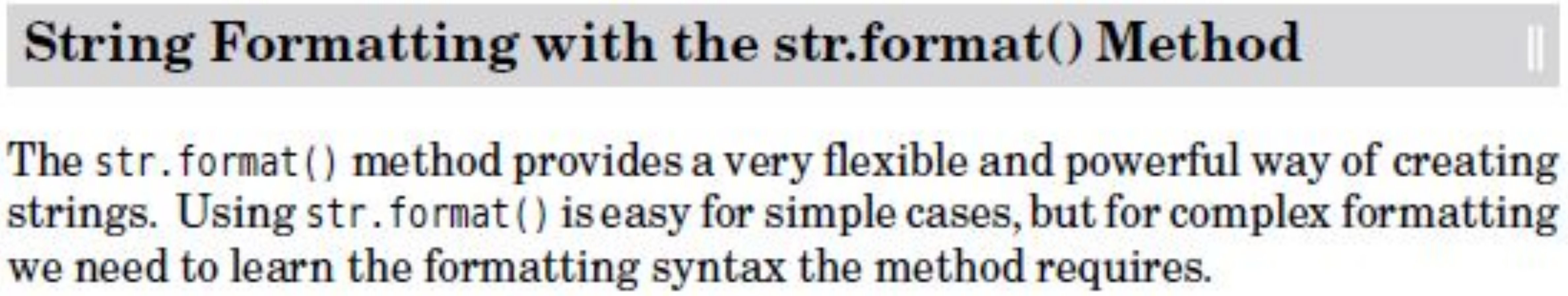


Field Names

A field name can be either an integer corresponding to one of the `str.format()` method's arguments, or the name of one of the method's keyword arguments. We discuss keyword arguments in Chapter 4, but they are not difficult, so we will provide a couple of examples here for completeness:

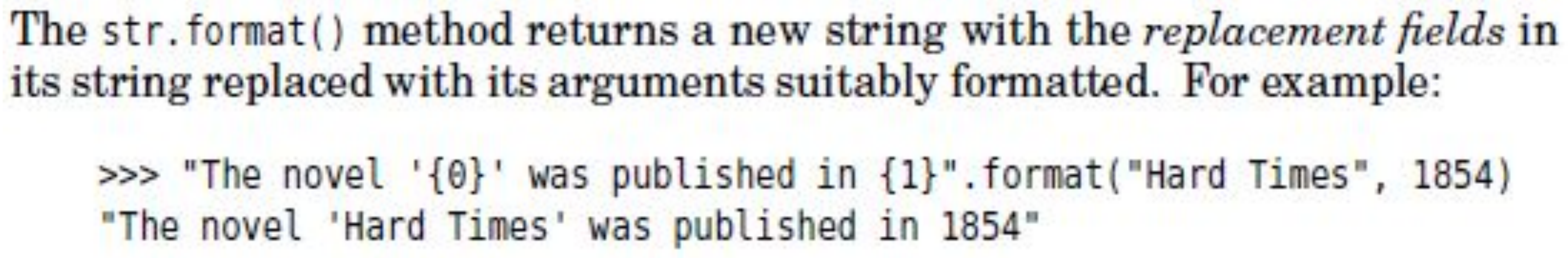
```
>>> "{who} turned {age} this year".format(who="She", age=88)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'The boy was 12 last week'
```

Introduction to Python



String Formatting with the str.format() Method

The `str.format()` method provides a very flexible and powerful way of creating strings. Using `str.format()` is easy for simple cases, but for complex formatting we need to learn the formatting syntax the method requires.



The `str.format()` method returns a new string with the *replacement fields* in its string replaced with its arguments suitably formatted. For example:

```
>>> "The novel '{0}' was published in {1}".format("Hard Times", 1854)
"The novel 'Hard Times' was published in 1854"
```

Introduction to Python

Formatted String Literals

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.
An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
>>>import math

>>>(f'The value of pi is approximately {math.pi:.3f}.')
```

```
The value of pi is approximately 3.142.
```

Introduction to Python

The String format() Method

Basic usage of the str.format() method looks like this:

```
>>>('We are the {} who say "{}!"'.format('knights', 'Ni'))

We are the knights who say "Ni!"
```

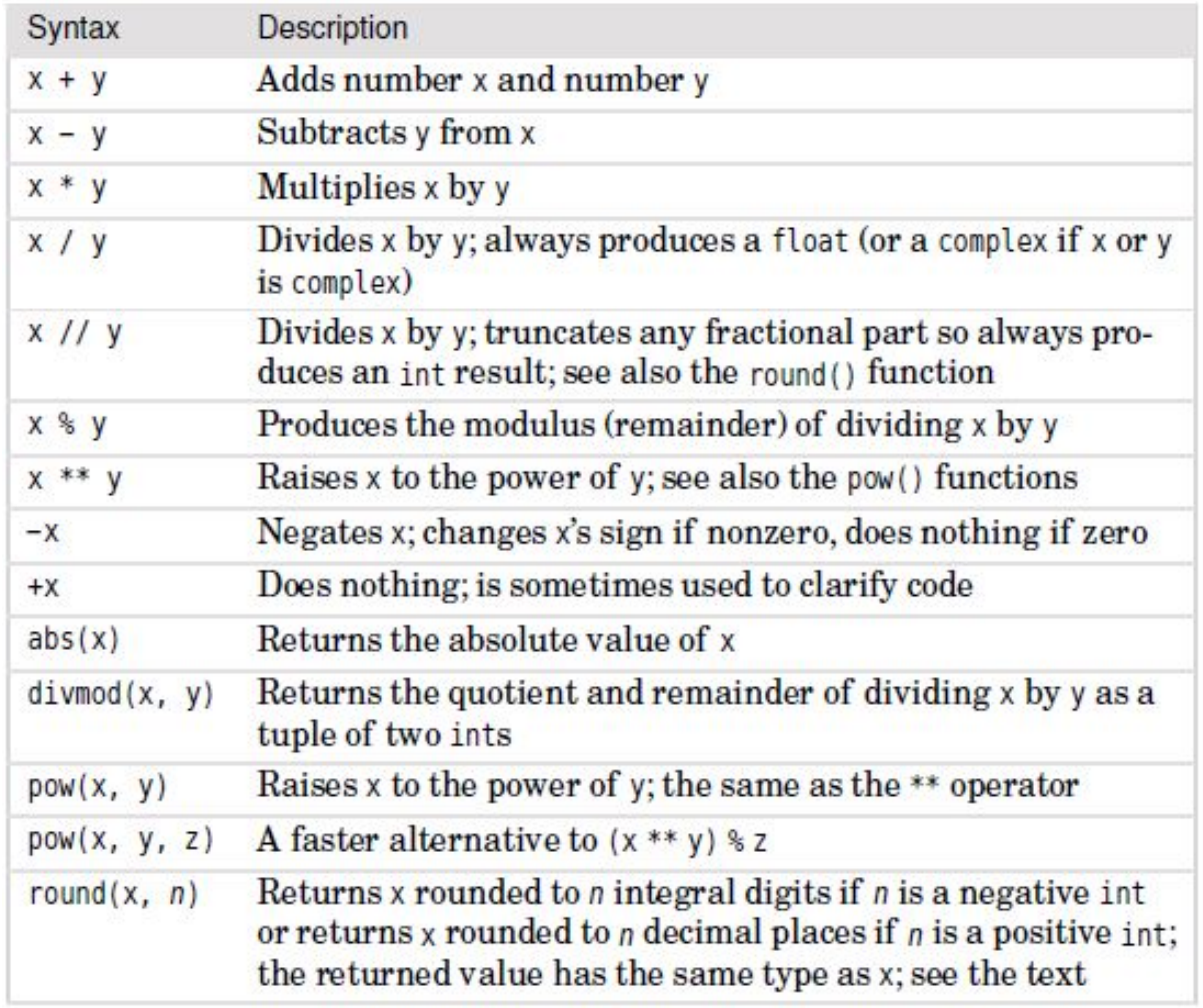
The brackets and characters within them (called format fields) are replaced with the objects passed into the str.format() method. A number in the brackets can be used to refer to the position of the object passed into the str.format() method.

```
>>>('{0} and {1}'.format('spam', 'eggs'))
spam and eggs


>>> ('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Introduction to Python

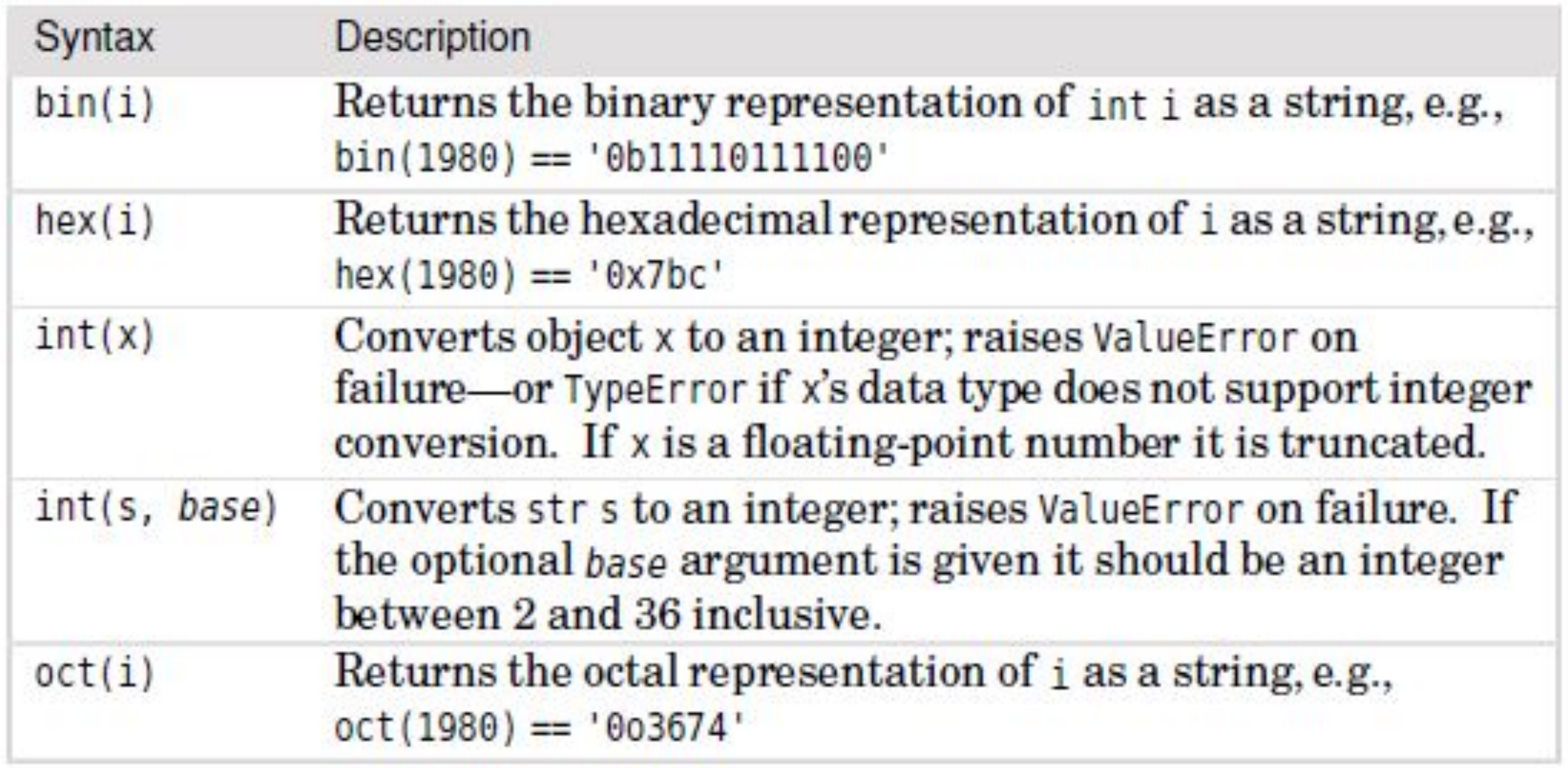
Integers *Numeric Operators and Functions*



Syntax	Description
`x + y`	Adds number `x` and number `y`
`x - y`	Subtracts `y` from `x`
`x * y`	Multiplies `x` by `y`
`x / y`	Divides `x` by `y`; always produces a `float` (or a `complex` if `x` or `y` is `complex`)
`x // y`	Divides `x` by `y`; truncates any fractional part so always produces an `int` result; see also the `round()` function
`x % y`	Produces the modulus (remainder) of dividing `x` by `y`
`x ** y`	Raises `x` to the power of `y`; see also the `pow()` functions
`-x`	Negates `x`; changes `x`'s sign if nonzero, does nothing if zero
`+x`	Does nothing; is sometimes used to clarify code
`abs(x)`	Returns the absolute value of `x`
`divmod(x, y)`	Returns the quotient and remainder of dividing `x` by `y` as a tuple of two `int`s
`pow(x, y)`	Raises `x` to the power of `y`; the same as the `**` operator
`pow(x, y, z)`	A faster alternative to `(x ** y) % z`
`round(x, n)`	Returns `x` rounded to *n* integral digits if *n* is a negative `int` or returns `x` rounded to *n* decimal places if *n* is a positive `int`; the returned value has the same type as `x`; see the text

Introduction to Python

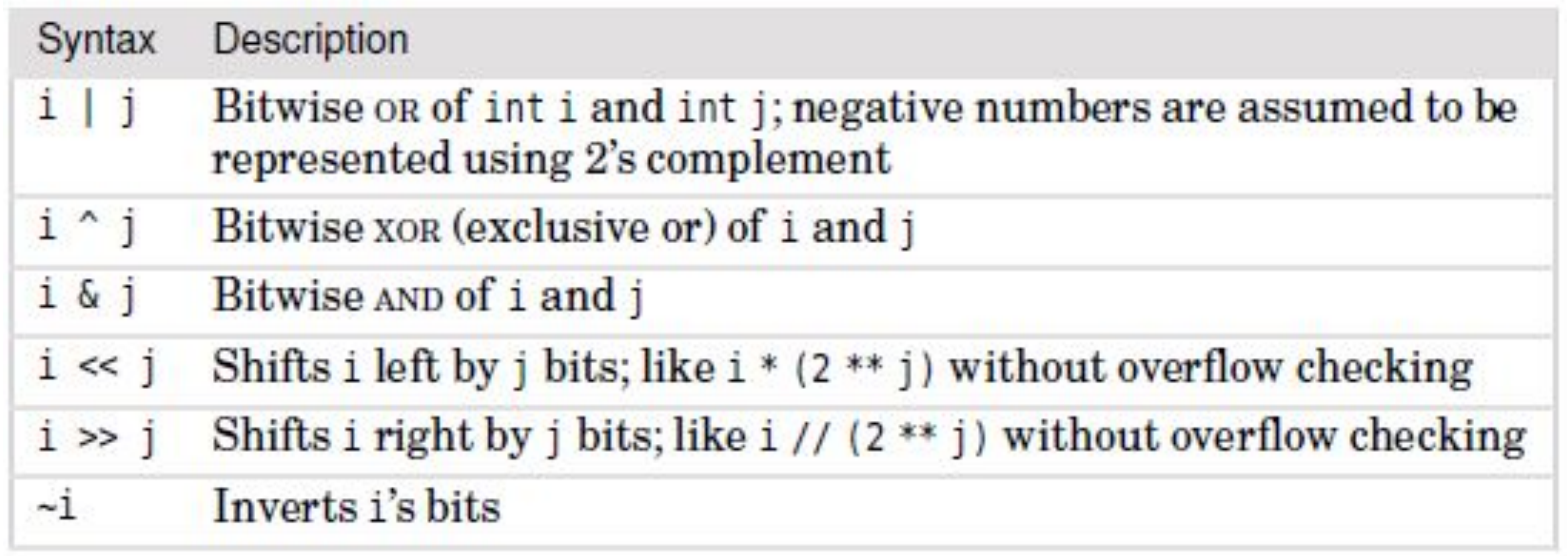
Integers *Integer Conversion Functions*



Syntax	Description
`bin(i)`	Returns the binary representation of `int` `i` as a string, e.g., `bin(1980) == '0b11110111100'`
`hex(i)`	Returns the hexadecimal representation of `i` as a string, e.g., `hex(1980) == '0x7bc'`
`int(x)`	Converts object `x` to an integer; raises `ValueError` on failure—or `TypeError` if `x`'s data type does not support integer conversion. If `x` is a floating-point number it is truncated.
`int(s, *base*)`	Converts `str` `s` to an integer; raises `ValueError` on failure. If the optional *`base`* argument is given it should be an integer between 2 and 36 inclusive.
`oct(i)`	Returns the octal representation of `i` as a string, e.g., `oct(1980) == '0o3674'`

Introduction to Python

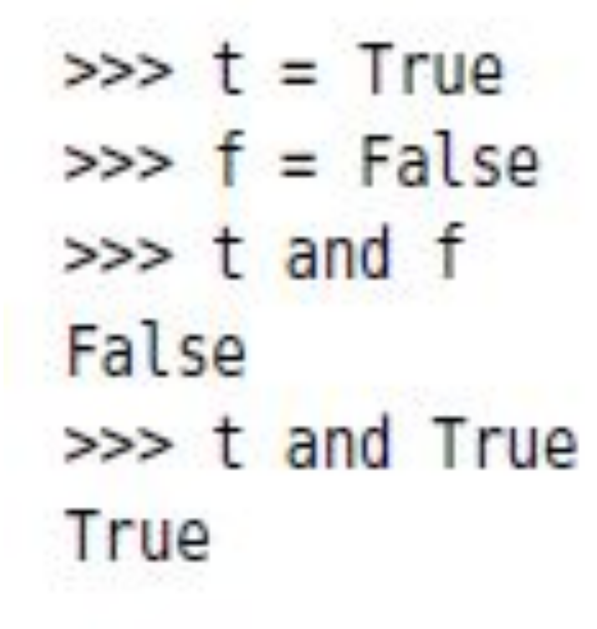
Integers *Integer Bitwise Operators*



Syntax	Description
`i | j`	Bitwise OR of `int` i and `int` j; negative numbers are assumed to be represented using 2's complement
`i ^ j`	Bitwise XOR (exclusive or) of i and j
`i & j`	Bitwise AND of i and j
`i << j`	Shifts i left by j bits; like `i * (2 ** j)` without overflow checking
`i >> j`	Shifts i right by j bits; like `i // (2 ** j)` without overflow checking
`~i`	Inverts i's bits

Introduction to Python

Booleans



```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Introduction to Python

Write a python program to take input registration number.

Perform string operation and find the year of joining, completion of the degree and print them.

If your branch start with B then 4 year course and if start with M then 5 year course.

Introduction to Python

Introduction to Python

Introduction to Python

Introduction to Python

References

Text Books:
Mark Summerfield, “Programming in Python 3”, 2nd Edition, Pearson Education, 2011.
Harvey M. Deitel, “Python – How to program” , Prentice Hall, 2002
Reference Books:
Michael Dawson, “Python Programming for the Absolute Beginner”, Third Edition, Cengage Learning, 2010.
Mark Lutz, “Learning Python”, 4th Edition, O’Reilly, 2000.

Online MOOC:

https://nptel.ac.in/courses/106/106/106106145/
https://nptel.ac.in/courses/106/106/106106182/

THANK-YOU