Python TAP (Theory, Application and Practice)

Samyar Modabber

2023-06-03

content

Introduction

In this book, I will teach you Python 3 from the ground up.

I will start with the basics and then move on to more advanced topics. I will also provide you with exercises and projects to help you practice what you have learned. I will also provide you with a reference guide to help you look up information quickly.

Part I

Basics of Python

1 Basic Concepts

1.1 Variables

A variable holds a value. You can store strings, numbers, or any data.

```
city = "Berlin"
population = 3850000
area = 891.8
is_capital = True
rivers=None
```

Python is **case-sensitive**. This means that variable names, function names, and identifiers with different capitalizations are treated as completely different

---

1.2 Built-in Constants

In Python, **built-in constants** are special predefined names that represent fixed values. They are always available and do not require an import.

None represents the absence of a value. This is useful when you want to define a variable before assigning a real value.

None is a singleton object: there is only one instance of **NoneType**.

True and **False** is two boolean constant.

1.2.1 Common Python Data Types

Here's a table of **common Python data types** with their descriptions and examples:

Type	Class Name	Description	Example
String	**str**	sequence of characters	**"hello"**, **'Python'**
Integer	**int**	Whole numbers	**42**, **-3**
Floating-point	**float**	Decimal numbers	**3.14**, **-0.01**
Boolean	**bool**	Logical values	**True**, **False**
None	**NoneType**	Represents no value	**None**

1.3 Comments

In Python, **comments** are used to explain the code and are **ignored during execution**.

- **Single-line comment**

```
# This is a single-line comment
```

- **Inline comment**

```
x = 5  # Store 5 in x
```

- **Multi-line comment**

```
"""
This is a multi-line comment
or documentation block.
"""
```

1.4 Display Output

In Python, **print()** is a built-in function used to display output on the screen (usually in the terminal or console). The basic syntax is

```
print(object1, object2, ..., sep=' ', end='\n')
```

1.4.1 Common Uses

- **Print one or multiple items or variable**

```
print("Hello, world!")  # Print one item
print("Age:", 25)       # Print multiple items
city = "Berlin"
print("City: ", city)   # Print variable
```

- **Change separator**

```
print("A", "B", "C", sep="-") # Output: A-B-C
```

- **Change end character**

```
print("Hello", end=" ")
print("World")
# Output: Hello World
```

1.4.2 Formatted output

1. Using f-string (Recommended)

```
print(f"{city} has {population:,} million people and an area of {area:.1f} km².")
```

2. Using `str.format()`

```
print("{} has {:,} million people and an area of {:.1f} km².".format(city, population, area)
```

3. Using % formatting (old style)

```
print("%s has %,d million people and an area of %.1f km²." % (city, population, area))
```

Output:

```
Berlin has 3.85 million people and an area of 891.8 km².
```

1.5 Check Data Type

The **type()** function in Python is used to check the **data type** of a variable or value.The Syntax is

```
type(object)
```

1.5.1 Examples

```
print(type(42))          # <class 'int'>
print(type(3.14))        # <class 'float'>
print(type("hello"))     # <class 'str'>
print(type(None))        # <class 'NoneType'>
print(type(True))        # <class 'bool'>
print(type(print))       # <class 'builtin_function_or_method'>
```

The **isinstance()** function in Python is used to **check if a value is an instance of a specific type or class**. The Syntax is

```
isinstance(object, <class_name>)
```

Examples

```
x = 42
s = "hello"
print(isinstance(x, int))     # True
print(isinstance(x, float))   # False
print(isinstance(s, str))     # True
print(isinstance(s, bool))    # False
```

1.6 Type Conversion

In Python, **Type Conversion** means converting a value from one data type to another. Two Types of Type Conversion are:

Type	Description
Implicit	Done automatically by Python
Explicit	Done manually using functions (casting)

1.6.1 1. Implicit Type Conversion

Python automatically converts types during expressions:

```
x = 5       # int
y = 2.0     # float
z = x + y   # int + float → float
print(z)    # 7.0
print(type(z))  # <class 'float'>
```

1.6.2 2. Explicit Type Conversion (Casting)

You can use built-in functions (class name):

Function	Converts to	Example
int()	Integer	int("5") → 5
float()	Float	float("3.14") → 3.14
str()	String	str(10) → "10"
bool()	Boolean	bool(0) → False

1.7 Get User Input

The **input()** function in Python is used to **take input from the user** as a **string**. The syntax is

```
variable = input("Prompt message")
```

- The message inside **input()** is optional and is shown to the user.
- The return value is always a **string** (`str`), even if the user types a number.

```
name = input("What is your name? ")
print("Hello, " , name)
```

1.7.1 Common Mistake

Always validate or cast carefully. The input should convert to integer by **int**.

```
# This will cause an error if input is not a number
age = int(input("Enter your age: "))
print(age + 5)
```

1.8 Practice Task

1.8.1 Practice Task 1

```python
# Define a constant for the value of Pi (used in circle area calculation)
PI = 3.14159

# Ask the user to enter their name and store it as a string
name = input("Enter your name: ")

# Ask the user to enter the radius, convert the input from string to float
radius = float(input("Enter the radius of a circle: "))

# Calculate the area of the circle using the formula: area =   * r^2
area = PI * radius * radius

# Greet the user by name
print("Hello", name)

# Display the calculated area of the circle
print("Circle area is:", area)
```

2 Conditions and Loops

2.1 Boolean Type

2.1.1 Introduction

In Python, **booleans** are a built-in data type that represent one of two values: **True** and **False**

These are the two built-in constant of the **bool** type.

```
type(True)   # <class 'bool'>
type(False)  # <class 'bool'>
```

Booleans are often used in:

- Comparisons
- Conditionals (**if**, **while**)
- Logical operations

2.1.2 Comparison Operators

In Python, **comparison operators** are used to compare values. These operators return **Boolean values**: **True** or **False**.

Operator	Meaning	Example	Result
==	Equal to	**3 == 3**	**True**

Operator	Meaning	Example	Result
!=	Not equal to	**4 != 5**	**True**
>	Greater than	**7 > 2**	**True**
<	Less than	**1 < 0**	**False**
>=	Greater than or equal to	**5 >= 5**	**True**
<=	Less than or equal to	**6 <= 3**	**False**

2.1.3 Example

```
x = 10
y = 5

print(x > y)       # True
print(x == y)      # False
```

You can also store boolean values in variables:

```
is_tall = True
is_ready = False
is_positive=30>0
```

2.1.4 Operator on Objects

Operator	Meaning	Example	Result
is	**Object identity**	**a is b**	**True** if same object
is not	Negated object identity	**a is not b**	**True** if not same object
in	Membership	**'a' in 'abc'**	**True**
not in	Negated membership	**'z' not in 'abc'**	**True**

Comparison with None

Use **is** and **is not** instead of == for **None** in order to test an variable is change to None or not. Because **None** is a singleton, and identity checks (**is**) are more precise than equality checks (==)

```
x = None
y = 2
print(x is None) # True
print(x is not None) # False
print(y is None) # True
```

Compare `is` vs ==

==	Compares **values**
is	Compares **identity** (memory address)

```
a = 1
b = 1
print(a == b)    # True (same contents)
print(a is b)    # False (different objects)
```

2.1.5 combination of statments

You can also combination of statments

Operator	Description	Example
and	Logical AND	**True and False** → **False**
or	Logical OR	**True or False** → **True**
not	Logical NOT	**not True** → **False**

Example Code

```python
x = 10
y = 20
z = 30
print(x == y and z>y)     # False
print(x != y or y<z)     # True
print(not x >= y)     # True
```

Chained Comparisons

Python supports **chaining** of comparisons:

```python
x = 5
print(1 < x < 10)     # True
print(1 < x and x < 10)  # Same result, more verbose
```

Boolean as a Subclass of Integer

```python
True == 1      # True
False == 0     # True
True + True    # 2
False + 3      # 3
```

But for readability, it's best to use **True** and **False** explicitly for logical operations rather than numeric 1 and 0.

2.2 if-elif-else: Making Decisions

Build-in keywords **if**, **elif** and **else** are used to **control the flow** of your program based on conditions.

2.2.1 if Statement

```
age = 18

if age >= 18:
    print("You are an adult")
```

In Python, **curly braces {} are not used** for blocks like in some other languages (e.g., C, Java, JavaScript). Instead, Python uses **colen** and **indentation**.

2.2.2 if-else Statement

```
age = 16

if age >= 18:
    print("Adult")
else:
    print("Minor")
```

2.2.3 if-elif-else Chain

```
score = 75

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

2.3 while - Loop Condition

The **while** loop keeps running **as long as** the condition is **True**.

```
count = 1

while count <= 5:
    print("Count is:", count)
    count += 1  # Same as count = count + 1
```

Output:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

2.3.1 Infinite Loop

If you forget to change the condition, the loop can run forever!

```
# WARNING: This loop never stops!
while True:
    print("Hello")  # Press Ctrl + C to stop in terminal
```

2.3.2 Controlling a loop

break — Stop the Loop Early

```
x = 1

while x <= 10:
    if x == 5:
        break  # Exit the loop when x is 5
    print(x)
    x += 1
```

Output:

```
1
2
3
4
```

2.3.3 continue — Skip One Iteration

```
x = 0

while x < 5:
    x += 1
    if x == 3:
        continue  # Skip the rest of the loop for x = 3
    print(x)
```

Output:

```
1
2
4
5
```

2.4 for - Loop

In Python, the **for** loop is used to iterate over a sequence (like a str that is a sequence of charechtors). Here's how it works:

```
for variable in iterable:
    # code block
```

2.4.1 Examples

1. For through a string

```
for letter in "hello":
    print(letter)
```

2.4.2 2. Build a reversed string

```
text = "hello"
reversed_text = ""

for char in text:
    reversed_text = char + reversed_text

print(reversed_text) # Output: olleh
```

---

3. for with break and continue

```
word = "Python"
for char in word:
    if char == "y":
        break
    print(char, end="_") # P_
```

```
word = "Python"
for char in word:
    if char == "y":
        continue
    print(char, end="_") # P_t_h_o_n_
```

2.5 Summary Table

Concept	Example	Description
Comparison	**x > y**, **x == y**	Compares values
Boolean	**True**, **False**	Logical values
if	**if x > 0:**	Run code if condition is True
elif	**elif x == 0:**	Check another condition
else	**else:**	Run if all above are False
while	Repeat while condition is true	**while x < 5:**
break	Exit the loop immediately	**if x == 5: break**
continue	Skip current loop and go to next one	**if x == 3: continue**

2.6 Practice Tasks

2.6.1 Practice Task 1

Write a program that:

1. Asks the user to enter a number
2. Checks if the number is: **Positive**, **Zero** or **Negative**

```
number = float(input("Enter a number: "))

if number > 0:
    print("The number is positive")
elif number == 0:
    print("The number is zero")
else:
    print("The number is negative")
```

2.6.2 Practice Task 2

Write a program that:

1. Asks the user to guess a secret number between 1 and 10
2. Repeats until the user guesses correctly

```
secret = 7
guess = None

while guess != secret:
    guess = int(input("Guess the number (1-10): "))
    if guess == secret:
        print("Correct!")
    else:
        print("Try again!")
```

2.6.3 Practice Task 3

Write a program that:

1. Asks the user to enter a number
2. Count the number of “o”s or “O”s in a given text

```
text = str(input("Enter a text: "))
count = 0

for char in text:
    if char == "o":
        count = count + 1
    elif char == "O":
        count = count + 1

print(f'Number of "o": {count}')
```

Practice Task 4

Write a program to:

1. Asks the user to enter a number
2. classify someone as **child, adult, or old** based on their age: If age is **less than 18**, print “child”. If age is **between 18 and 64** (inclusive of 18), print “adult”. Otherwise (65 and above), print “old”.

```python
age = int(input("Enter your age: "))

if age < 18:
    print("You are a child.")
elif 18 <= age < 65:
    print("You are an adult.")
else:
    print("You are old.")
```

3 Simple Data Types

3.1 Numeric Type

In Python, **numeric types** are built-in data types used to store and manipulate numbers. There are three main numeric types:

Type	Description	Example
int	Integer numbers	**10**, **-5**, **0**
float	Floating-point (decimal) numbers	**3.14**, **-0.01**
complex	Complex numbers (real + imag part)	**2 + 3j**, **-1j**

3.1.1 int —Integer

- Whole numbers (no decimal point)
- Unlimited precision (arbitrary size)

```
x = 42
print(type(x))  # <class 'int'>

big = 10**100   # Very large integer
print(type(big)) # <class 'int'>
```

3.1.2 float — Floating-Point

- Numbers with a decimal point
- Internally based on IEEE 754 double-precision (64-bit)

```
pi = 3.14159
print(type(pi))  # <class 'float'>
div = 1 / 3
print(div)  # 0.333...
```

Special float values:

```
float('inf')     # ∞
float('-inf')    # -∞
float('nan')     # Not a Number
```

3.1.3 complex —Complex Numbers

- Numbers with **real** and **imaginary** parts
- Written as **a + bj** (use **j**, not **i**)

```
z = 2 + 3j
print(type(z))     # <class 'complex'>
print(z.real)      # 2.0
print(z.imag)      # 3.0
```

3.2 Text Sequence Type

In Python, the **str** type represents **textual data** — it's one of the most commonly used built-in types. In fact in a **str** is an **immutable sequence of Unicode characters**.

```
name = "Sam"
print(type(name))  # <class 'str'>
```

Creating Strings

```
s1 = 'hello'
s2 = "world"
```

Multiline Strings

```
poem = """Roses are red,
Violets are blue,
Python is awesome,
And so are you."""
```

3.2.1 String Formatting

We can embed variabels in string and print it.

```
name = "Sam"
age = 30

# .format()
print("My name is {} and I am {}".format(name, age))

# f-string (Python 3.6+)
print(f"My name is {name} and I am {age}")
```

3.2.2 Immutability

Strings are **immutable**. It means you can not modify (delete,add,update) any part of a string:

```
s = "hello"
s[0] = "H"  # TypeError
```

To modify a string, you must create a new one:

```
s = "hello"
s = "H" + s[1:]  # 'Hello'
```

3.3 Operations

3.3.1 String Operators

Here's a **basic operation on strings** in Python, organized in a readable table format:

Operation	Description	Example	Output
+	Concatenation – combines two strings	**'Hello' + ' World'**	**'Hello World'**
*	Repetition – repeats a string	**'Ha' * 3**	**'HaHaHa'**
len(s)	Returns the length of the string	**len('hello')**	**5**
s[i]	Indexing – gets character at index i	**'hello'[1]**	**'e'**
s[start:end]	Slicing – substring from **start** to **end-1**	**'hello'[1:4]**	**'ell'**
in	Membership test – checks if a substring exists	**'lo' in 'hello'**	**True**
not in	Negated membership test	**'z' not in 'hello'**	**True**

Notes:

- **Indexing** starts at **0**, so **'hello'[1]** is the second character, **'e'**.
- **Slicing** does **not** include the character at the **end** index.
- The **in** and **not in** operations are useful in conditionals and loops.

3.3.2 Numeric Operators

Operators Supported by All Numeric Types:

- Arithmetic: +, -, *****, /, //, %, "'
- Comparison: ==, !=, <, >, <=, >=

```
a = 5
b = 2
print(a + b)    # 7
print(a / b)    # 2.5
print(a // b)   # 2 (floor division)
```

Here's a clean and complete table of **numeric operations** in Python, including their **description**, **example result**, and **notes**, based on the built-in types: **int**, **float**, and **complex**.

Numeric Operations Table in Python

Operation	Result	Notes
-x	Negation of **x**	–
+x	**x** unchanged	Unary plus
x + y	Sum of **x** and **y**	Works for all numeric types
x - y	Difference of **x** and **y**	–
x * y	Product of **x** and **y**	–
x ** y or **pow(x, y)**	**x** to the power **y**	Exponentiation (5)
x / y	Quotient of **x** and **y** (float)	Always returns float
x // y	Floored quotient of **x** and **y**	Discards fractional part
x % y	Remainder of **x / y**	Result has same sign as **y**
divmod(x, y)	Returns **(x // y, x % y)**	Tuple result -> (x//y,x%y)
abs(x)	Absolute value of **x**	Also works for complex numbers
round(x,d)	Rounds **x** to the nearest integer	e.g., **round(2.337,2)** -> 2.34
int(x)	Remove decimal part of **x**	e.g., **int(-2.93)** -> -2
c.conjugate()	Conjugate of complex number **c**	e.g., **(2+3j).conjugate() = 2-3j**

Notes:

1. // performs **floor division**, e.g. **5 // 2 = 2**, **-5 // 2 = -3**
2. **%** uses Python's **modulus rule**: the result has the **same sign as the divisor** (y)
3. **int(3.8)** → **3**; **int('7')** → **7**; **int('abc')** → raises error
4. **float('3.14')** → **3.14**
5. **pow(2, 3)** → **8**; works like **2 3; can also do modular exponentiation:** pow(2, 3, 5)** → **3**
6. **int()**, **float()**, and **complex()** are constructors for their respective types
7. You can find official documentation on https://docs.python.org

4 Structured Data Types

Structured data types let you **store multiple values** together (like collections or containers).

4.1 Immutable Structured Types

Once created, these cannot be changed (no item modification, addition, or deletion).

Type	Description	Example
str	Text (sequence of characters)	**"hello"**
tuple	Ordered, fixed-size collection	**(1, 2, 3)**
range	Sequence of numbers	**range(5)**

Examples:

```
# String
text = "hello"
print(text[0])      # Output: h

# Tuple
point = (3, 4)
print(point[1])     # Output: 4

# Range
nums = range(3)
print(list(nums))   # Output: [0, 1, 2]
```

4.2 Mutable Structured Types

These can be **changed** after creation (add, remove, or update items).

Type	Description	Example
list	Ordered, changeable collection	**[1, 2, 3]**
set	Unordered, unique items only	**{1, 2, 3}**
dict	Key-value pairs	**{‘name’: ‘Ali’}**

Examples:

```
# List
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)  # ['apple', 'banana', 'cherry']

# Set
nums = {1, 2}
nums.add(3)
print(nums)    # {1, 2, 3}

# Dictionary
person = {"name": "Ali", "age": 25}
person["age"] = 26
print(person)  # {'name': 'Ali', 'age': 26}
```

4.3 Compare Table

Category	Type	Mutable?	Ordered?	Allows Duplicates?
Sequence	**list**			
Mapping	**dict**		(3.7+)	(keys must be unique)
Set	**set**			
Sequence	**tuple**			
Sequence	**range**			
Text	**str**			

4.4 Practice Task

Write a program that:

1. Creates a list of numbers: **[1, 2, 3]**
2. Adds **4** to the list
3. Converts it to a tuple
4. Prints both the list and the tuple

```python
nums = [1, 2, 3]
nums.append(4)
t = tuple(nums)

print("List:", nums)
print("Tuple:", t)
```

Part II

Practice

5 Basic and String

5.1 Basic

5.1.1 Hello World!

```python
#Print Hello
print('Hello world')

#Case sensetive
# Print('Hello world')
```

```
Hello world
```

input

```python
name=input("Give me your name: ")
print("Hello,", name)
```

```
Hello,
```

variable

```python
message='Hello world!'
# print(message)
# type(message)
# len(message)
```

```python
a=b=c='Hello'
print(a,b,c)
```

```
Hello Hello Hello
```

```
first_name='Sara'

#Wrong syntax
# first-name ='Sara'
# 5_name ='Sara'
# first name ='Sara'
```

Single qoutes and duble qoutes

```
message1="He's my friend."

message2='His name is "Milad".'
#  He's "Milad"
message3='He\'s an "Artist".\n New line'

print(message1)
print(message2)
print(message3)
```

```
He's my friend.
His name is "Milad".
He's an "Artist".
 New line
```

Multi line

```
message4="""Hello Milad
I am intereted to learn Python.
Could you teach me?
Sincerely,
Sohrab
"""
print(message4)
```

```
Hello Milad
I am intereted to learn Python.
Could you teach me?
Sincerely,
Sohrab
```

Indexing and Slicing

```
message5='Hi. Where are you nowdays?'

## one element
print(message5[0])


# Interval
print(message5[0:4])
print(message5[:4])
print(message5[4:])
print(message5[19:25])
print(message5[0:25:2])

## Negetive
print(message5[-1])
print(message5[-7])

print(message5[8:-7])
print(message5[-2:2:-1])
print(message5[::-1])
```

```
H
Hi.
Hi.
Where are you nowdays?
owdays
H.Weeaeyunwas
?
o
e are you n
syadwon uoy era erehW
?syadwon uoy era erehW .iH
```

lower, UPPER

```
message6='thanks dear Saam.'
print(message6.lower())
print(message6.upper())
print(message6.capitalize())
print(message6.count('s'))
print(message6.count('aa'))

print(message6.find('Saam'))
print(message6.find('Sam'))
```

```
thanks dear saam.
THANKS DEAR SAAM.
Thanks dear saam.
1
1
12
-1
```

replace

```
message7='I am intereted to learn Python.'
edited=message7.replace('e','_')
print(edited)

print(message7)

message7=message7.replace('e','_')
print(message7)
```

```
I am int_r_t_d to l_arn Python.
I am intereted to learn Python.
I am int_r_t_d to l_arn Python.
```

```
message7='I am intereted to learn Python.'
edited=message7.replace('e','_',2)
print(edited)
```

```
I am int_r_ted to learn Python.
```

concat

```
name='Ramin'
welcome='Welcome, dear'

message8=welcome+ ' ' + name+'!'
print(message8)
```

```
Welcome, dear Ramin!
```

format and f

```
name='Armin'
unread_messages=23

print('Dear {}, you have {} unreaded message(s).'.format(name,unread_messages))

print(f'Dear {name}, you have {unread_messages} unreaded message(s).') #python 3.6+

print(f'Dear {name.upper()}, you have {unread_messages:3d} unreaded message(s).') #python 3.
```

```
Dear Armin, you have 23 unreaded message(s).
Dear Armin, you have 23 unreaded message(s).
Dear ARMIN, you have  23 unreaded message(s).
```

dir and help

```
message10='bye'

# print(dir(message10))
# print(help(str))
print(help(str.find))
```

```
Help on method_descriptor:

find(self, sub[, start[, end]], /) unbound builtins.str method
    Return the lowest index in S where substring sub is found, such that sub is contained wi

    Optional arguments start and end are interpreted as in slice notation.
    Return -1 on failure.

None
```

5.2 Methods

In Python strings are **immutable**. This means that for instance the following assignment is not legal:

```
s="text" s[0] = "a"    # This is not legal in Python
```

Because of the immutability of the strings, the string methods work by returning a value; they don't have any side-effects.

In the rest of this section we briefly describe several of these methods. The methods are here divided into five groups.

1. Classification of strings:

All the following methods will take no parameters and return a truth value. An empty string will always result in False.

```
s=' All the following Methods.'

# s.isalpha() #True if all characters are letters
# s.isdigit() #True if all characters are digits
# s.isalnum() #True if all characters are letters or digits

# s.islower() #True if contains letters, and all are lowercase
# s.isupper() #True if contains letters, and all are uppercase
#s.isspace() #True if all characters are whitespace
#s.istitle() #True if uppercase in the beginning of word, elsewhere lowercase
```

```
False
```

2. String transformations:

The following methods do conversions between lower and uppercase characters in the string. All these methods return a new string.

```
s=' All the following Methods.'
print('main srting:',s)

sl=s.lower() #Change all letters to lowercase
print('lower:',sl)

su=s.upper() #Change all letters to uppercase
print('upper:',su)
```

```python
sc=s.capitalize() #Change all letters to capitalcase
print('capitalize:',sc)

st=s.title() #Change to titlecase
print('title:',st)

ss=s.swapcase() #Change all uppercase letters to lowercase, and vice versa
print('swapcase:',ss)
```

```
main srting:  All the following Methods.
lower:  all the following methods.
upper:  ALL THE FOLLOWING METHODS.
capitalize:  all the following methods.
title:  All The Following Methods.
swapcase:  aLL THE FOLLOWING mETHODS.
```

3. Searching for substrings:

All the following methods get the wanted substring as the parameter, except the replace method, which also gets the replacing string as a parameter

```python
s=' All the following methods'
substr='ll'
m=s.count(substr) #Counts the number of occurences of a substring
print(f'sc:{m}')

sf=s.find(substr) #Finds index of the first occurence of a substring, or -1
sr=s.rfind(substr) #Finds index of the last occurence of a substring, or -1
print(f'sf:{sf} sr:{sr}')

##ValueError
si=s.index(substr) #Like find, except ValueError is raised if not found
sri=s.rindex(substr) #Like rfind, except ValueError is raised if not found
print(f'si:{si} sri:{sri}')

target='All'
start=s.startswith(target) #Returns True if string starts with a given substring
end=s.endswith(target) #Returns True if string ends with a given substring
print(f'start:{start} end:{end}')

replacement='-'
```

```
sn=s.replace(substr, replacement) #Returns a string where occurences of one string are repla
print(f'sn:{sn}')
```

```
sc:2
sf:2 sr:11
si:2 sri:11
start:False end:False
sn: A- the fo-owing methods
```

4. Trimming and adjusting

```
s='  Removes   leading and    '
x='s'
s.strip() #Removes leading and trailing whitespace by default, or characters found in string
#s.lstrip(x) #Same as strip but only leading characters are removed
#s.rstrip(x) #Same as strip but only trailing characters are removed
n=20
#s.ljust(n) #Left justifies string inside a field of length n
#s.rjust(n) #Right justifies string inside a field of length n
s.center(n) #Centers string inside a field of length n
```

5. Joining and splitting:

The join(seq) method joins the strings of the sequence seq. The string itself is used as a delimitter. An example:

```
allStr="*".join(["abc", "def", "ghi"])
print(allStr)
```

```
abc*def*ghi
```

```
s='method joins the strings of the sequence'
sp=s.split()
print(sp)
```

```
['method', 'joins', 'the', 'strings', 'of', 'the', 'sequence']
```

6 Integers and Float

6.1 Numbers

types

```
num1=19
num2=3.14
num3=2+3j
print(type(num1))
print(type(num2))
print(type(num3))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

basic operators

```
num1=19
num2=7


print('Addition:',num1+num2)
print('Subtraction:',num1-num2)

print('Multiplication:',num1*num2)
print('Exponent:',num1**num2)

print('Division:',num1/num2)
print('Floor Division:',num1//num2)
print('Modulus:',num1%num2)


type(num1)
```

```
Addition: 26
Subtraction: 12
Multiplication: 133
Exponent: 893871739
Division: 2.7142857142857144
Floor Division: 2
Modulus: 5
```

```
int
```

```python
num=19
# num=num%2

num-=2 ## It can be + - * / % num=num%2
print(num)
```

```
17
```

```python
num1=-3
print(abs(num1))

num2=3.14
print(int(num2))

num3=16
print(num3**(1/2))
```

```
3
3
4.0
```

```python
num1=19.19
print(round(num1))
num1=19.99
print(round(num1))

num1=19.19
print(round(num1,1))
num1=19.99
print(round(num1,1))
```

```
num1=-19.19
print(round(num1))
num1=-19.99
print(round(num1))
```

```
19
20
19.2
20.0
-19
-20
```

6.2 Logic

```
p=True
q=False

print(p and q)
print(p or q)
print(not q)

print(p is True)
print(q is not True)

print(None is False)
print(0 is False)

type(print)
```

```
False
True
True
True
True
False
False
```

```
<>:12: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:12: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-4-43b64229a9fc>:12: SyntaxWarning: "is" with a literal. Did you mean "=="?
  print(0 is False)
```

```
builtin_function_or_method
```

6.3 Compare numbers

```
num1=19
num2=7

print(f'{num1} is equal {num2}?', num1==num2)
print(f'{num1} is not equal {num2}?', num1!=num2)

print(f'{num1} is greater than {num2}?', num1>num2)
print(f'{num1} is less than {num2}?', num1<num2)

print(f'{num1} is greater than or equal {num2}?', num1>=num2)
print(f'{num1} is less than or equal {num2}?', num1<=num2)

result= num1==num3
print('Result:',result)
```

```
19 is equal 7? False
19 is not equal 7? True
19 is greater than 7? True
19 is less than 7? False
19 is greater than or equal 7? True
19 is less than or equal 7? False
Result: False
```

```
not num1!=num2
```

```
False
```

6.4 Convert

```python
print(int(-2.8))
print(float(2))
print(int("123"))
print(bool(-2), bool(0))  # Zero is interpreted as False
print(str(234))
```

```
-2
2.0
123
True False
234
```

```python
num1='19'
num2='7'

# print(num1+num2)

print(int(num1)+int(num2))
```

```
26
```

```python
True or True and False #  == True or (True and False)
```

```
True
```

```python
(True or True) and False
```

```
False
```

```
True
```

```python
not False and False # == (not False) and True
```

```
False
```

```python
not (False and False)
```

```
True
```

7 Data structures

7.1 Introduction

Data type * int * complex * float * boolian

Data structures

The main data structures in Python divided to two categories: * A. Sequences: strings, list, tuples

- B. Non-sequences: dictionaries, sets

7.2 A. Sequences (List, tuples, and strings) have several commonalities:

1. Their length can be queried with the `len` function.
2. Thet are `immutable`.
3. They can be concatenated with the `+ operator`.
4. They repeated with the `* operator`.
5. Since they are `ordered`, we can refer to the elements by integers using the `indexing` notation.

```
s='A list contains arbitrary number of elements.'

# len(s)
# s[0]='a'
# s+s
# 'Arash '*10
# s[3]
s[-1]
```

```
'.'
```

7.3 A. Sequences: List

A list contains arbitrary number of elements (even zero) that are stored in sequential order. The elements are separated by commas and written between brackets. The elements don't need to be of the same type. An example of a list with four values:

```python
mylist1=[2, 100, "hello", 1.0]
print(mylist1)
print(id(mylist1))

mylist2=[2, 100, "hello", 1.0]
print(id(mylist2))


# print(mylist1==mylist2)
print(mylist1 is mylist2) # print(id(mylist1() == id(mylist2))
```

```
[2, 100, 'hello', 1.0]
2371224743872
2371224757440
False
```

Indexing and Slicing

```python
courses=['Calcules','Physics','Computer','Statistic','Algebra']
# print(courses)

# print(courses[2])
# print(courses[-1])

# print(courses[2:4])
# print(courses[2:])
# print(courses[:2])

print(courses[0:4:2])
# print(courses[-1:0:-1])

# print(courses[::2])
print(courses[1:-2])
```

```
['Calcules', 'Computer']
['Physics', 'Computer']
```

Modifying

```
courses=['Calcules','Physics','Computer','Statistic','Algebra']
# print(courses)
#Modifying element of a lists
courses[0]='Calcules1'
# print(courses)
# #Modifying any slice of a lists
courses[0:2]=['Calcules1','Calcules2','Physics1','Physics2']
# print(courses)
courses.remove('Calcules2')
courses.remove('Physics2')
# print(courses)
courses.append('Analysis')
# print(courses)
courses.insert(4,'Logic')
# print(courses)
newCourses=['Geometry','ODE','PDE']
courses.extend(newCourses)
# print(courses)
poped=courses.pop(-2) # pop()==pop(-1)
print(courses)
print(poped)
```

```
['Calcules1', 'Physics1', 'Computer', 'Statistic', 'Logic', 'Algebra', 'Analysis', 'Geometry
ODE
```

```
courses=['Calcules','Physics','Computer','Statistic','Algebra']

delete=courses.pop(1)
print(courses)
```

```
['Calcules', 'Computer', 'Statistic', 'Algebra']
```

```
delete
```

```
'Physics'
```

```
courses=['Calcules','Physics','Computer','Statistic','Algebra']

# courses.reverse()
# print(courses)

# courses.sort()
# print(courses)

# courses.sort(reverse=True)
# print(courses)

# after_sorted=sorted(courses)
# print(after_sorted)
# print(courses)

# marks=[-5,-4,3,2,6]
# simple_sorted=sorted(marks)
# print(simple_sorted)

abs_sorted=sorted(marks, key=abs)
print(abs_sorted)
```

```
[2, 3, -4, -5, 6]
```

```
scores=[12, 17, 14, 10, 3, 9, 20, 18]
print(min(scores))
print(max(scores))
print(sum(scores))
print(len(scores))
```

```
3
20
103
8
```

```
courses=['Calcules1','Physics1','Computer','Statistic','Algebra']

# print(courses.index('Statistic'))
# print('Computer' in courses)
# tostring=' '.join(courses)
# print(tostring)

my_st='A list contains arbitrary number of elements'
words_list=my_st.split(' ')
print(words_list)
```

```
['A', 'list', 'contains', 'arbitrary', 'number', 'of', 'elements']
```

Range Function

Trivial lists can be tedious to write: [0,1,2,3,4,5,6]. The function range creates numeric ranges automatically.

- Then end value is not included in the sequence.
- consumes less memory than the corresponding list.

This is because in a list all the elements are stored in the memory, whereas the range generates the requested elements only when needed. For example, when the for loop asks for the next element from the range at each iteration, only a single element from the range exists in memory at the same time. This makes a big difference when using large ranges, like range(1000000).

```
range1=range(7)
# print(type(range1)) # Note that L is not a list!
# print(list(range1))

# range2=range(3,7)
# print(list(range2))

range3=range(0,20,3)
print(list(range3))
```

```
[0, 3, 6, 9, 12, 15, 18]
```

```
list(range1)
int('123')
```

```
123
```

7.4 A. Sequences: Tuple

A tuple is fixed length, immutable, and ordered container. Elements of tuple are separated by commas and written between parentheses. Examples of tuples:

```
singleton=(3,)              # a singleton
pair=(1,3)              # a pair
triple=(1, "hello", 1.0); # a triple
type(singleton)
```

```
tuple
```

```
triple[2]
```

```
1.0
```

Note the difference between (3) and (3,). Because the parentheses can also be used to group expressions, the first one defines an integer, but the second one defines a tuple with single element. As we can see, both lists and tuples can contain values of different type.

We can also modify a list by using mutating methods of the list class, namely the methods append, extend, insert, remove, pop, reverse, and sort.

Note that we cannot perform these modifications on tuples or strings since they are immutable

7.5 B. Non-Sequences: Sets

A set is a dynamic, unordered container.

```
semester1={'Calcules1','Physics','Computer','Statistic'}
semester2={'Calcules2','Computer','Algebra'}

# print(semester1)
# print('Algebra' in semester1)

# print(semester1.intersection(semester2))
# print(semester1.difference(semester2))
# print(semester1.union(semester2))
```

```
empty_set=set() # Not {}
type(empty_set)
```

```
set
```

7.6 B. Non-Sequences: Dictionaries

A dictionary is a dynamic, unordered container.

Instead of using integers to access the elements of the container, the dictionary uses **keys** to access the stored values.

The dictionary can be created by listing the comma separated key-value pairs in braces. Keys and values are separated by a colon.

A tuple (key,value) is called an **item** of the dictionary.

```
student1={'name':'Sara',
          'age':23,
          'student_id':2020121110,
          'courses':{'Calcules2','Computer','Algebra'}
         }

student2=dict([
    ('name', 'Danial'),
    ('age', 22),
    ('student_id', 2019121002)
])

student3=dict(
    name='Arman',
    age=18,
    courses=set()
);


print(student1['name'])
print(student1.pop('age'))
print(student1.get('student_id'))

#print(student1['phone']) # Error
```

```python
print(student1.get('phone'))
print(student1.get('phone','I can Not Found'))

#Add a key-value
student1['phone']='222-2222-2222'
print(student1['phone'])

student2.clear()
```

```
Sara
23
2020121110
None
I can Not Found
222-2222-2222
```

```python
student1={'name':'Sara',
          'age':23,
          'student_id':2020121110,
          'courses':{'Calcules2','Computer','Algebra'}
         }
# print(student1)

# student1.update({'age':24,'courses':{'Logic','Linear Algebra'}})
# print(student1)

# del student1['courses']
# print(student1)


# print(len(student1))
# print(student1.keys())
# print(student1.values())
# print(student1.items())

# for key,value in student1.items():
#     print(key,value)

student1.clear()
print(student1)
```

```
{}
```

```
student4={}
print(student4)
print(type(student4))
```

```
{}
<class 'dict'>
```

8 Conditions and Loops

8.1 if, elif, else

```
# a='sara' #23
# b='saara' #22
a =[2,3,5,8]
b= [2,3,7]

if a==b:
    print(f'{a} is equal {b}.')
elif a>b:
    print(f'{a} is greater than {b}.')
elif a<b:
    print(f'{a} is less than {b}.')
else:
    print('Non of theme')
```

```
[2, 3, 5, 8] is less than [2, 3, 7].
```

8.1.1 Logic operators

```
user='Admin'
logged_in=False

if user=='Admin' and logged_in:
    print ('Admin Page')
elif not logged_in:
    print('Login Page')
```

```
Login Page
```

8.1.2 False Value

- False
- None
- Zero
- Empty structure: ’ ’, [], (), {}

```
if None or 0 or [] or () or {} or '':
    print('One of them is True')
else:
    print('All of them are False')
```

```
All of them are False
```

8.2 For

```
courses=['Calcules','Physics','Computer','Statistic','Algebra']

for c in courses:
    print(c)
```

```
Calcules
Physics
Computer
Statistic
Algebra
```

```
courses=['Calcules','Physics','Computer','Statistic','Algebra']

for _ in courses:
    print('*')
```

```
*
*
*
*
*
```

```python
list(range(7))
```

```
[0, 1, 2, 3, 4, 5, 6]
```

```python
for i in range(7):
    if i%2==0:
        print(f'{i} is Even.')
    else:
        print(f'{i} is Odd.')
```

```
0 is Even.
1 is Odd.
2 is Even.
3 is Odd.
4 is Even.
5 is Odd.
6 is Even.
```

```python
limit=3

for i in range(1,7):
    if i<limit:
        print(f'Your password is not correct. {i}/{limit}.')
    else:
        print(f'You input incorrected password {i} times. Your card is blocked.')
        break
```

```
Your password is not correct. 1/3.
Your password is not correct. 2/3.
You input incorrected password 3 times. Your card is blocked.
```

```python
for i in range(20):
    if i%3==0:
        continue
    print(i)
```

```
1
2
4
```

```
5
7
8
10
11
13
14
16
17
19
```

```
for i in range(3):
    for j in 'abc':
        print(i,j)
```

```
0 a
0 b
0 c
1 a
1 b
1 c
2 a
2 b
2 c
```

8.3 while

```
x = 0

while x<=10:
    x+=1
    print(x)
```

```
1
2
3
4
5
```

```
6
7
8
9
10
11
```

```python
x
```

```
11
```

```python
u=[2 ,3,6,8,5]
v=[-1,5,0,6,7]

s=0
n=len(u)

for i in range(n):
    s+=u[i]*v[i]

s
```

```
96
```

```python
u=[2 ,3,6,8,5,4,2,5,2]
v=[-1,5,0,6,7,-1,8,6,5]

s=0
n=len(u)

if len(u)!=len(v):
    print(f'dim is not equal: {len(u)} , {len(v)}')
else:
    for i in range(n):
        s+=u[i]*v[i]

s
```

```
148
```

```
u=[2 ,3,6,8,5,4,2,5,2]
v=[-1,5,0,6,7,-1,8,6,5]

s=0
n=len(u)

if len(u)==len(v):
    for i in range(n):
        s+=u[i]*v[i]
else:
    print(f'dim is not equal: {len(u)} , {len(v)}')


s
```

```
148
```

```
def inner_product(a,b):
    m=len(a)
    n=len(b)
    s=0
    if m==n :
        for i in range(m):
            s+=a[i]*b[i]
    else:
        return 'Dimentions are not equal'
    return s
```

```
inner_product(u,v)
```

```
148
```

```
u1=[2,3,6]
u2=[0,5,9]
inner_product(u1,u2)
```

```
69
```

```
u1=[2,3,6,3]
u2=[0,5,9]
inner_product(u1,u2)
```

```
'Dimentions are not equal'
```

```
a=[1,2,3,4,5,6,7,8,9]



# 1 2 3
# 4 5 6
# 7 8 9
```

```
list(range(1,26))
```

```
[1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25]
```

9 Function

9.1 Define a Function

```python
def hello_func():
    '''Doc string: This function print a string'''
    print('Hello User')
hello_func()
```

```
Hello User
```

```python
hello_func()
```

```
Hello User
```

```python
hello_func?
```

```python
list?
```

```python
def login_msg(name):
    print(f'Hello {name}, you login now!')
```

```python
login_msg('Samyar')
```

```
Hello Samyar, you login now!
```

```python
login_msg('Arash')
```

```
Hello Arash, you login now!
```

```
def login_msg(user):
    return f'Hello {user}, you login now!'
    print('Hi')
```

```
msg=login_msg('Samyar')
# print(msg)
msg
```

```
'Hello Samyar, you login now!'
```

```
def user_age(name,age):
    return f'{name} is {age} years old.'
```

```
# print(user_age('Parviz',28))
```

```
user_age(28,'Parviz')
```

```
'28 is Parviz years old.'
```

```
user_age(age=28,name='Parviz')
```

```
'Parviz is 28 years old.'
```

```
user_age(28)
```

```
TypeError: user_age() missing 1 required positional argument: 'age'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-f65f0252b944> in <module>
----> 1 user_age(28)

TypeError: user_age() missing 1 required positional argument: 'age'
```

```
def user_age(name='User name',age=19):
    return f'{name} is {age} years old.'
print(user_age())
```

```
User name is 19 years old.
```

```python
user_age('Reza',32)
```

```
'Reza is 32 years old.'
```

```python
def user_age(name='User',age=19):
    return f'{name} is {age} years old.'
```

```python
user_age(age=18)
```

```
'User is 18 years old.'
```

9.2 args, kwargs

Pass ARRAY and DICTIONARY as arguments of a function.

```python
def student_courses(*courses):
    print(courses)
student_courses('Calcules1','Algebra1','Logic')
```

```
('Calcules1', 'Algebra1', 'Logic')
```

```python
def student_details(**details):
    print(details)
student_details(neme='Parviz',age=28, is_active=True)
```

```
{'neme': 'Parviz', 'age': 28, 'is_active': True}
```

```python
def student_info(*courses,**details):
    print(courses)
    print(details)
student_courses('Calcules1','Algebra1','Logic',neme='Parviz',age=28)
```

```
('Calcules1', 'Algebra1', 'Logic')
{'neme': 'Parviz', 'age': 28}
```

```python
def student_info(*courses,**details):
    print(courses)
    print(details)

courses=['Calcules1', 'Algebra1', 'Logic']
details={'neme': 'Parviz', 'age': 28}
student_courses(*courses,**details)
```

```
('Calcules1', 'Algebra1', 'Logic')
{'neme': 'Parviz', 'age': 28}
```

9.3 Function in Function

```python
def inc(x):
    return x + 1


def dec(x):
    return x - 1


def operate(func, x):
    result = func(x)
    return result

operate(inc,3)
```

```
4
```

```python
def list_courses(courses):
    print('The courses:')
    for course in courses:
        print(course)

def student_des(*courses,**details):
    for key,value in details.items():
        print(f'{key}:{value}')
    list_courses(courses)
```

```
courses=['Calcules1', 'Algebra1', 'Logic']
details={'neme': 'Parviz', 'age': 28}
student_des(*courses,**details)
```

```
neme:Parviz
age:28
The courses:
Calcules1
Algebra1
Logic
```

9.4 Scope (LEGB)

Local, Enclosing, Global, Built-in

```
x='global x'
def test():
    y='local y'
    #print(y)
    print(x)
test()
#print(y) ##Error
```

```
global x
```

```
x='global x'
def test():
    x='local x'
    print(x)
test()
print(x)
```

```
local x
global x
```

```python
x='global x'
def test():
    global x
    x='local x'
    print(x)
test()
print(x)
```

```
local x
local x
```

```python
def test(z):
    print(z)
test('local z')
#print(z) ##Eror
```

```
local z
```

```python
import builtins
print(dir(builtins))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
```

```python
m=abs(-10)
print(m)
```

```
10
```

```python
# You can overwrite bult-in functions
def abs(x):
    if x>0:
        return x
    else:
        return 0

m=abs(-10)
print(m)
```

```
0
```

9.5 Enclosing

```python
x='global x'
def outer():
    #x='outer x'
    def inner():
        #x='inner x'
        print(x)
    inner()

outer()
```

```
global x
```

9.6 Return a Function

```python
def outer():
    message='Hello'
    def inner():
        print(message)
    return inner()

outer()
```

```
Hello
```

```python
def outer():
    message='Hello'
    def inner():
        print(message)
    return inner # not Exe

my_func = outer()
my_func()
```

```
Hello
```

```python
def outer(message):
    def inner():
        print(message)
    return inner # not Exe

hi_func = outer('Hi')
bye_func= outer('Bye')
hi_func()
bye_func()
```

```
Hi
Bye
```

9.7 Decorator Function

A decorator takes in a function, adds some functionality and returns it.

```python
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
```

```python
def ordinary():
    print("I am ordinary")
pretty = make_pretty(ordinary)
pretty()
```

```
I got decorated
I am ordinary
```

```python
@make_pretty
def ordinary():
    print("I am ordinary")
ordinary()
```

```
I got decorated
I am ordinary
```

9.7.1 Example

```
def divide(a, b):
    return a/b
#divide(2,0)
```

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return
        return func(a, b)
    return inner
```

```
@smart_divide
def divide(a, b):
    print(a/b)
divide(2,0)
```

```
I am going to divide 2 and 0
Whoops! cannot divide
```

9.7.2 Multiple decorators can be chained in Python.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner
```

```python

```

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

```python
@star
@percent
def printer(msg):
    print(msg)
#printer = star(percent(printer)) we should use this if we dont use decorations
printer("Hello")
```

```python
@percent
@star
def printer(msg):
    print(msg)

printer("Hello")
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

References