SSTIA python workshop I

```
>>> print("Hello World!")
Hello World!
```

Python Hands-on Tutorial

Online Courses

- CS61A: Structure and Interpretation of Computer Programs (UCB)
  - 课程网站：https://inst.eecs.berkeley.edu/~cs61a/su20/
  - 课程视频: 参见课程网站链接
  - 课程教材：https://www.composingprograms.com/
  - 课程教材中文翻译：https://composingprograms.netlify.app/
- CS50P Introduction to Programming with Python
  - 课程网站：2022
  - 课程视频：2022

Books

- Python Book Guide and List: Python语言相关的编程书籍推荐列表

Tutorials

- Python Official Tutorials
  - The Python Tutorial — Python 3.12.0 documentation
- GeeksforGeeks: Python Tutorials
  - Python Tutorial | Learn Python Programming (geeksforgeeks.org)

Basic Syntax

Data Type

Overview

Python has several built-in data types. Here are some of them:

- **Text Type**: `str`
- **Numeric Types**: `int`, `float`, `complex`
- **Sequence Types**: `list`, `tuple`, `range`
- **Mapping Type**: `dict`
- **Set Types**: `set`, `frozenset`
- **Boolean Type**: `bool`
- **Binary Types**: `bytes`, `bytearray`, `memoryview`
- **None Type**: `NoneType`

You can get the data type of any object by using the `type()` function. For example, to get the data type of the variable `x`, you can use `type(x)`. If you want to specify the data type, you can use the following constructor functions: `str()`, `int()`, `float()`, `complex()`, `list()`, `tuple()`, `range()`, `dict()`, `set()`, `frozenset()`, `bool()`, `bytes()`, `bytearray()`, `memoryview()`.

```
# DataType Output: str
x = "Hello World"
```

```python
# DataType Output: int
x = 50

# DataType Output: float
x = 60.5

# DataType Output: complex
x = 3j

# DataType Output: list
x = ["geeks", "for", "geeks"]

# DataType Output: tuple
x = ("geeks", "for", "geeks")

# DataType Output: range
x = range(10)

# DataType Output: dict
x = {"name": "Suraj", "age": 24}

# DataType Output: set
x = {"geeks", "for", "geeks"}

# DataType Output: frozenset
x = frozenset({"geeks", "for", "geeks"})

# DataType Output: bool
x = True

# DataType Output: bytes
x = b"Geeks"

# DataType Output: bytearray
x = bytearray(4)

# DataType Output: memoryview
x = memoryview(bytes(6))

# DataType Output: NoneType
x = None
```

Text Type `str`

Mapping Type: `dict`

Sequential Type: `tuple`

In Python, the `tuple` type is a built-in data type used to represent an ordered and immutable collection of elements. Tuples are defined by enclosing comma-separated values within parentheses `()`. Once a tuple is created, its elements cannot be modified, added, or removed.

Creating a Tuple:

You can create a tuple by enclosing elements in parentheses. Here are a few examples:

```python
# Creating a tuple with integers
my_tuple = (1, 2, 3)
print(my_tuple)
```

```python
# Creating a tuple with mixed data types
mixed_tuple = ("apple", 42, True, 3.14)
print(mixed_tuple)
```

```python
# Creating an empty tuple
empty_tuple = ()
print(empty_tuple)
```

```python
# Creating a tuple with a single element (note the trailing comma)
single_element_tuple = (42,)
print(single_element_tuple)
```

Accessing Elements in a Tuple:

You can access elements in a tuple using indexing, similar to lists. The index starts from 0 for the first element.

```python
my_tuple = (1, 2, 3, 4, 5)

# Accessing the first element
first_element = my_tuple[0]
print("First Element:", first_element)

# Accessing the third element
third_element = my_tuple[2]
print("Third Element:", third_element)
```

Tuple Unpacking:

Tuple unpacking allows you to assign the elements of a tuple to individual variables.

```python
coordinates = (5, 10)
x, y = coordinates

print("x:", x)
print("y:", y)
```

Common use: Unpack tuple from the return values of function. Example:

```
def operate(a, b):
    add = a + b
    sub = a - b
    return (add, sub)

x = 10
y = 5
add_result, sub_result = operate(x, y)
```

Immutability:

One key characteristic of tuples is their immutability. Once a tuple is created, you cannot change its elements. This makes tuples suitable for situations where the order and immutability of elements are important.

Operator

Arithmetic Operators

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Here is an example showing how different Arithmetic Operators in Python work:

```
# Examples of Arithmetic Operator
a = 9
b = 4

# Addition of numbers
add = a + b

# Subtraction of numbers
sub = a - b

# Multiplication of number
mul = a * b

# Modulo of both number
mod = a % b

# Power
p = a ** b
```

```
# print results
print(add)
print(sub)
print(mul)
print(mod)
print(p)
```

Output:

```
13
5
36
1
6561
```

Comparison Operators

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

Let's see an example of Comparison Operators in Python.

```
# Examples of Relational Operators
a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
```

```
print(a <= b)
```

Output

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

Logical Operators

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	not x

The following code shows how to implement Logical Operators in Python:

```
# Examples of Logical Operator
a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output

```
False
True
False
```

Bitwise Operators in Python

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

IO

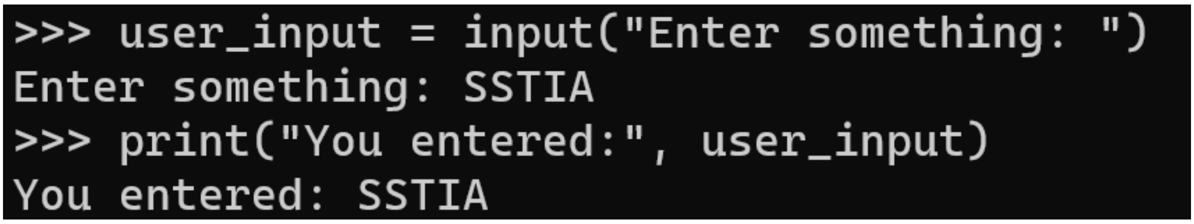
In Python, input and output operations (I/O) are commonly performed using the built-in `input()` function for receiving input from the user and the `print()` function for displaying output. Additionally, Python provides various modules for more advanced I/O operations. Let's cover the basics first:

Basic Input and Output:

Input:

The `input()` function is used to take user input. It reads a line from the user and returns it as a string. You can convert the input to other data types as needed.

```
# Basic Input Example
user_input = input("Enter something: ")
print("You entered:", user_input)
```



```
>>> user_input = input("Enter something: ")
Enter something: SSTIA
>>> print("You entered:", user_input)
You entered: SSTIA
```

Take multiple input:

Using split() method :

Example:

```
# Python program showing how to
# multiple input using split

# taking two inputs at a time
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)

# taking three inputs at a time
x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)

# taking two inputs at a time
a, b = input("Enter two values: ").split()
print("First number is {} and second number is {}".format(a, b))

# taking multiple inputs at a time
# and type casting using list() function
x = list(map(int, input("Enter multiple values: ").split()))
print("List of students: ", x)
```

Output:

```
# Python program showing how to
# multiple input using split

# taking two inputs at a time
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)

# taking three inputs at a time
x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)

# taking two inputs at a time
a, b = input("Enter two values: ").split()
print("First number is {} and second number is {}".format(a, b))

# taking multiple inputs at a time
# and type casting using list() function
x = list(map(int, input("Enter multiple values: ").split()))
print("List of students: ", x)
```

Output:

The `print()` function is used to display output to the console. You can pass multiple values separated by commas to `print()`.

```
# Basic Output Example
name = "John"
age = 25
print("Name:", name, "Age:", age)
```

File I/O:

Python provides built-in functions for working with files: `open()`, `read()`, `write()`, and `close()`. Here's a simple example:

```
# File I/O Example
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, this is a sample text.")

# Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    print("File Content:", content)
```

In the above example, the `with` statement is used to automatically close the file when the block is exited.

Control Flow

Python If Else

Syntax:

```
if (condition):
<Tab>statement
elif (condition):
<Tab>statement
.
.
else:
<Tab>statement
```

Example:

```
# Python program to illustrate if-elif-else ladder

i = 20
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Python For Loops

For Loops Syntax

```
for var in iterable:
    # statements
```

zip

Python While Loop

Syntax:

```
while expression:
    statement(s)
```

Functions

Types of Functions in Python

- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

Creating a Function in Python

- syntax1:

```
def function_name(parameter):
    # body of the function
    return expression
```

- syntax2:

```
def function_name(parameter: data_type) -> return_type:
    """Docstring"""
    # body of the function
    return expression
```

Example:

```
def add(num1, num2):
    num3 = num1 + num2
    return num3
```

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2

    return num3

# Driver code
num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

Default Arguments

```python
# Python program to demonstrate
# default arguments
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)


# Driver code (We call myFun() with only
# argument)
myFun(10)
```

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

```python
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
    print(firstname, lastname)


# Keyword arguments
student(firstname='Geeks', lastname='Practice')
student(lastname='Practice', firstname='Geeks')
```

Docstring

```python
# A simple Python function to check
# whether x is even or odd


def evenOdd(x):
    """Function to check if the number is even or odd"""

    if (x % 2 == 0):
        print("even")
    else:
        print("odd")


# Driver code to call the function
print(evenOdd.__doc__)
```

Pass by Reference and Pass by Value

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is the same as reference passing in C++.

```
# Here x is a new reference to same list lst
def myFun(x):
    x[0] = 20


# Driver Code (Note that lst is modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

```
[20, 11, 12, 13, 14, 15]
```

When we pass a reference and change the received reference to something else, the connection between the passed and received parameter is broken. For example, consider the below program as follows:

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = [20, 30, 40]


# Driver Code (Note that lst is not modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

Another example demonstrates that the reference link is broken if we assign a new value (inside the function).

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = 20


# Driver Code (Note that x is not modified
# after function call.
x = 10
myFun(x)
print(x)
```

Output:

```
10
```

Exercise

- Try to guess the output of the following code.

```python
def swap(x, y):
    temp = x
    x = y
    y = temp


# Driver code
x = 2
y = 3
swap(x, y)
print(x)
print(y)
```

Class

Import Module

2 types of modules:

- Built-in Modules
- Self-written Modules

Module && Header files

在Python中的`import`语句和C语言中的头文件（header file）有一些相似之处，但也存在一些显著的区别。让我们来比较它们并提供一些简单的例子：

Python中的`import`语句：

在Python中，`import`语句用于引入模块，以便在代码中使用模块中定义的函数、类等。一个Python模块通常对应于一个文件，文件名即为模块名，包含了一些可复用的代码。

例子：

假设有一个名为`math_operations.py`的Python文件，其中包含了一些数学操作的函数：

```python
# math_operations.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y
```

然后在另一个文件中，你可以使用`import`语句引入这个模块并调用其中的函数：

```
# main.py
import math_operations

result_add = math_operations.add(5, 3)
result_subtract = math_operations.subtract(8, 2)

print(result_add)       # 输出: 8
print(result_subtract)  # 输出: 6
```

C语言中的头文件:

在C语言中，头文件通常包含了函数声明、宏定义等信息，可以在多个源代码文件中共享这些信息。头文件的内容会在编译时被包含到源代码中，以便确保所有文件中使用的函数、变量等都能正确识别。

例子:

假设有一个名为`math_operations.h`的头文件，其中包含了一些数学操作的函数声明:

```
// math_operations.h
#ifndef MATH_OPERATIONS_H
#define MATH_OPERATIONS_H

int add(int x, int y);
int subtract(int x, int y);

#endif
```

然后在另一个文件中，你可以使用`#include`指令引入这个头文件，并使用其中声明的函数:

```
// main.c
#include <stdio.h>
#include "math_operations.h"

int main() {
    int result_add = add(5, 3);
    int result_subtract = subtract(8, 2);

    printf("%d\n", result_add);       // 输出: 8
    printf("%d\n", result_subtract);  // 输出: 6

    return 0;
}
```

总体而言，Python的`import`和C语言的头文件都是为了组织和模块化代码，使代码更易于维护和复用。然而，它们的实现方式和语法有很大的不同。

Built-in Library

Let's provide examples of calling built-in libraries in both Python and C.

Python Example - Using the `math` Module:

Python has a built-in `math` module that provides mathematical functions. Here's an example:

```python
# Python Example - Using the math module
import math

# Calculate the square root of 25
sqrt_result = math.sqrt(25)

# Calculate the cosine of 45 degrees
cos_result = math.cos(math.radians(45))

print("Square Root:", sqrt_result)
print("Cosine of 45 degrees:", cos_result)
```

In this example, we import the `math` module and use its functions `sqrt` and `cos` to perform mathematical calculations.

C Example - Using the Standard Library (`math.h` for output):

Here's an example:

```c
// C Example - Using the math.h library
#include <stdio.h>
#include <math.h>

int main() {
    // Calculate the square root of 25
    double sqrt_result = sqrt(25.0);

    // Calculate the cosine of 45 degrees
    double cos_result = cos(45.0 * M_PI / 180.0);

    printf("Square Root: %.2f\n", sqrt_result);
    printf("Cosine of 45 degrees: %.2f\n", cos_result);

    return 0;
}
```

Import Python Standard Library Module

The Python standard library contains well over **200** modules. We can import a module according to our needs.

Suppose we want to get the value of `pi`, first we import the math module and use `math.pi`. For example,

```python
# import standard math module
import math

# use math.pi to get value of pi
print("The value of pi is", math.pi)
```

Output

```
The value of pi is 3.141592653589793
```

Python import with Renaming

In Python, we can also import a module by renaming it. For example,

```
# import module by renaming it
import math as m

print(m.pi)

# Output: 3.141592653589793
```

Here, We have renamed the `math` module as `m`. This can save us typing time in some cases.

Note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, and `m.pi` is the correct implementation.

Python from...import statement

We can import specific names from a module without importing the module as a whole. For example,

```
# import only pi from math module
from math import pi

print(pi)

# Output: 3.141592653589793
```

Here, we imported only the `pi` attribute from the `math` module.

Import all names

In Python, we can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math
from math import *

print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Exercise

- Check The Python standard library, and use module `operator` to concatenate 2 list a, b.

```
import operator

a = ["SSTIA", "Department of Technology"]
b = ["Have FUN with Python!!!"]

c = None
####################################################################
```

```
# Choose the member in the operator module to concatenate 2 list a and b,
# and store it in c.
# Theoretically, the answer should be
# ["SSTIA", "Department of Technology", "Have FUN with Python!!!"]
# ##################################################################
# Replace "pass" statement with your code
pass
#####################################################################
#                          END OF YOUR CODE                         #
#####################################################################
print(c)
```