**Python Fundamental1**

Overview

In this handout, we will cover the basics of Python statements, variables, identifiers, and the commonly used data types: integers, floating-point numbers, strings, and booleans. Understanding these data types is essential for performing arithmetic operations, handling textual data, and making logical decisions in programs. Additionally, we will explore fundamental control structures, including for and while loops, which allow for repeated execution of code blocks, as well as if, else, and elif statements for conditional decision-making. By mastering these basics, you will gain the foundational skills needed to write and understand simple Python programs.

## Statements, Variables, and Identifiers

A **statement** is a command that tells the program to do something.

# rhiH:i::i&:i+eiii'iii·fiiifH♦

= 12

This **statement** creates the variable x and assigns it the value 12 using the assignment operator =. Here's another example:



This **statement** adds the numbers 1 and 2 using the addition operator+. Let's look at one more statement:

# M::i+iii+i:HiM:i'i::H&+♦

> 3

This **statement** checks if 5 is greater than 3 using the comparison operator>.

Programs store and use data, which is kept in the computer's memory. A Python **statement** often contains one or more **variables. A variable** holds a value that you can use later in your program. In simple terms, a **variable** is a name that points to a value stored in the computer's memory.

1These notes and examples adapt the references listed at the end. They are compiled to fit the scope of this specific course.

**assignment of different values to different variables**

city\_name = "Washington DC" average\_temperature = 18.3

num\_of\_parks = 24 is\_coastal\_city = **False**

In this snippet (small piece of code), city\_name, average\_temperature, num\_of\_parks, and is\_coastal\_city are all **variables.** We assign values to these variables using the = operator. Each line here is an assign­ ment **statement.** You can later use these variables in your code to retrieve their values and update them by assigning new values. Notice that each of these variables stores different types of data. While city \_name stores a text, average\_temperature and num\_of\_parks store numerical values, whereas is\_coastal\_city stores a True/False kind of data.

**Note 1** : Python is case-sensitive! This means city \_name and City \_Name are not the same.

**Note 2:** In all the above statements, there's a space on both sides of the= sign. This is recommended by the **style guide for python code (PEP** 8), which makes your code easier to read. You can read more about the **PEP 8 style guide** on the website:

[https://www.python.org/dev/peps/pep-0008/](http://www.python.org/dev/peps/pep-0008/)

As we go through the course, you are expected to follow these rules.

An **identifier** is a name (like city\_name or num\_of\_parks in the examples) that you create to name **variables, functions, classes,** etc.

Identifiers must follow these rules:

* Identifiers can be made up of letters (a-z, A-Z), digits (0-9), and underscores(\_). Space is not allowed.
* Identifiers cannot start with a digit. For example, 1day is invalid, but day1 is valid.
* Identifiers can be as long as you want.
* You cannot use special symbols like ! , @, #, -, etc. The dash - is used as the minus operator.
* **Keywords** cannot be used as identifiers. Keywords (such as False, if, lambda, and import) are reserved words in Python that have special meanings and cannot be used as names for variables or functions. 2

Warning

Python has several built-in functions such as print(), type(), sum(), len (), min(), max(), and id(), among others. You will become familiar with these functions as you progress. Although it is technically possible to use these names as variable names, doing so is considered very poor programming practice and is strongly discouraged.

2For a full list of keywords, please refer to https: //docs. python. org/3/ref erence/lexical\_analysis .html#keywords. Alternatively, you can run

import keyword keyword.kwlist

inside a Jupyter Lab cell. Yet, another alternative is to open the Command Prompt from Anaconda Navigator and run

python help() keywords

1. **print O Function**

A **function** is a pre-written block of code that performs a specific task. You can call a function by writing its name followed by parentheses O. Data given to a function inside the parentheses O is called an **argument.**

The print O function displays output on the screen.

print O **function example**

interest= "Data Analysis" print(interest)

This code yields:

Data Analysis

In this example, interest is the argument passed to the print() function, and print() displays the value of this argument on the computer screen.

The print O function can accept more than one argument separated by a comma , .

###### print O function with more than one argument

city\_name = "Washington DC"

print("Hi,", "Welcome to", city\_name, ".")

This code yields:

Hi, Welcome to Washington DC.

In this example, city \_name is a **string3,** which is a sequence of characters enclosed in double quotes ".

Here is another example

**another** print O **example**

num\_1 = 5

num\_2 = 6

result= num 1 \* num\_2

print(num\_1, "\*", num\_2, "=" result)

This code yields: 5 \* 6 = 30

**3string** is python's text type and we will discuss it shortly.



print O is a built-in (ready-to-use) function that displays the output on the computer screen.

## Python Data Types

In Python, we can store different kinds of data in a variable. Different data types can perform different tasks. In the examples above, we have seen some of Python's most common four data types:

* my\_name is a **string** (str),
* hourly\_wage is a **floating-point number** (float),
* number\_of\_kids is an **integer** (int), and
* is\_employed is a **boolean** (bool).

These types are often referred to as Python's **basic** data types because they store single values. The following shows some of the most commonly used data structures in Python:

**Built-in**

**Data Types**

I

I I

-

-

**str**

(Text type) -

example: 'DC'

-

**bool**

(Boolean type)

example: True, False

**dictionary**

A collection of key: value pairs inside { )

example: {'Milo': 100,

James': 20 l

**set**

A set of items inside {)

Example: { 2, 4, 10)

**tuples**

A collection of items inside ( )

example: (2, 4, -9)

**list**

A collection of items inside (]

example: [2, -4, 'DC']

**Complex**

**Basic**

**int**

(Numeric Integer type)

example: 3, -7

**float**

( Numeric float type)

example: 3. 14, 0.2

Figure 1: Most common data types

We will learn basic data types first before diving into complex ones. It is noteworthy that Python also has a data type called complex for handling complex numbers, which are written with a j as the imaginary part. We will not use this data type in this class.

##### int: Python's Numerical Integer Type

int is Python's built-in type for storing integer numbers. An integer is a whole number without a decimal point. The following example assigns an int to a variable:

# hiH:ii:I-IIMii'iii-fii!fHa

umber\_1 = 15

This statement assigns the value 15 to the variable number\_1 using the assignment operator=.

In Python 3, there is no limit to how large an integer can be (except by the memory available on your computer). For example:

**testing** int **limit**

X = 9999999999999999999999999999999999999999999999999999999999999

y = X + 1

y

This code will yield: 10000000000000000000000000000000000000000000000000000000000000

You can check the type of a value or a variable using Python's built-in type O function. Here's an example:

**'Hi:,.,i®iii:iiiHiiii'i:MM:h**

**r**

X = 15

type(x)

This will return: int



type() is a built-in function that returns the data type of a value or variable.4

##### float: Python's Numerical Floating Point Type

float is Python's data type for floating-point numbers (numbers with a decimal point). A float represents real numbers and is written with a decimal point separating the integer and fractional parts. For example, -30.6, 25.0, 10.79, 8e3, and 3.9e128 are all floating-point numbers. (Reminder: 8e3 = 8.0 x 103 = 8.0 x 1000 = 8000.0).

# t'Hil"iiM!i!:lfridl'!:iiiM:►

z = 6.7

type(z)

This yields: float

4a more accurate definition is: *type is a built-in function that returns the type of an object or variable.* We will learn the concept of Python objects later.

The maximum value of a float in Python is approximately 1.8e308; any number greater than this will return the string inf (which stands for *infinity).* Similarly, the minimum value is about -1.8e308; numbers smaller than this are indicated by -inf (standing for *-infinity).5* Let's check this:

# IMi~~i+iii::iiiiliiM!fri♦~~

rint (1. 75e308)

This yields:

1.75e+308

However, if we try a number beyond the maximum:

# IMi~~i+iii::iiiiliiM!fri♦~~

rint (1. 8e308)

This yields: inf

* 1. **Arithmetic on** int **and** float

We often need to perform basic arithmetic operations for data analysis. The table below summarizes the arith­ metic operators in Python.

|  |  |  |  |
| --- | --- | --- | --- |
| **Python Operation** | **Arithmetic Operator** | **Algebraic Expression** | **Python Expression** |
| Addition | + | *f+7* | f + 7 |
| Subtraction | - | *b-c* | b-c |
| Multiplication | \* | *b·m* | **b\*m** |
| Exponentiation | \*\* | *xY* | **X** \*\* **y** |
| True division | *I* | *3:.*  *y* | *xly* |
| Floor division | *II* | I I | *XII* y |
| Remainder (modulo) | % | *r* mods | r%s |

Table 1: Arithmetic operations on numerical values

**A few hints:**

* You can use the exponentiation operator to find the square root of a number! Remember that,17 = 47°·5.
* Floor division (/I) divides a numerator by a denominator, yielding the largest integer that is not greater than the result. Hence, 10 *II* 4 = 2 and -10 *II* 4 = -3. (This is because -10 *I* 4 = -2.5, and the largest integer that

51f you are interested in where the 308 limit comes from, please check the https: //en. wikipedia. org/wiki/Double-precision\_ floating-point\_format on Wikipedia.

is not greater than -2.5 is -3.)

* Dividing by zero using / or// will result in a **ZeroDivisionError.**

#### Assignment Operators

In Python, you can use assignment operators to assign values to variables. Beyond the basic = operator, Python also supports **augmented assignment operators.** These operators allow you to perform a calculation and assign the result to the variable in a single step.

For example, instead of writing x = x + 3, you can use x += 3, which adds 3 to the value of x and assigns the result back to x. This is both simpler and more readable.

Here are the most common augmented assignment operators:

|  |  |  |  |
| --- | --- | --- | --- |
| **Algebraic Operator** | **Python Operator** | **Description** | **Python Example** |
|  | = | Assign a value | x=5 |
| + | += | Add and assign | X += 3 |
|  | -= | Subtract and assign | X -= 7 |
| X | \*= | Multiply and assign |  |
|  | /= | Divide and assign | X /= 2 |

Table 2: Augmented assignment operators

In the table, the +=, -=, \*=, and /= operators all work in a similar way: they perform an operation (addition, subtraction, multiplication, or division) on the variable and then assign the result back to the same variable. This makes your code more concise and easier to read.

#### str: Python's Text Data Type

* + 1. str **Basics**

str is Python's text type. A string, or str, is a sequence of characters (symbols). Each letter in the English language is a character, but characters are not limited to just English letters! Any combination of characters forms a string. For example:

* + - * 'stream'
      * "stream"
      * "Hello there!"
      * "!!!"
      * 'q@'

###### "m"

are all strings.

A string in Python can contain as many characters as you want; the only limit is your computer's resources. You can surround a Python str with either single quotes' or double quotes". There is no difference between the two (e.g., 'stream' is the same as "stream"). Even though you can use either type of quotes, it's a good practice to be consistent.

You can display a str using Python's print O function. The print O function will not show the quotes around the string.

# rifHUfiiiii:\·\

**Print("Build for Whats Next.")**

This yields:

Build for Whats Next.

However:

# riM'iiiifili:HIH+

rint('Build for What's Next.')

yields a **SyntaxError.** Do you know why?

You can also begin and end a str with triple double quotes (""") or triple single quotes ('"). The Python style guide recommends using triple double quotes. This can be used to create:

1. Long, multi-line strings
2. Strings containing single or double quotes
3. docstrings, which are used to document programs (we'll learn more about this later). Try this:

**printing strings with single quotes and double quotes**

print("""Build For What's Next.""")

print("""Build For What's Next. 'This' Is How "We" Roll. 111111)

When a str contains a single quote, use double quotes around the string, and vice versa.

Alternatively, you can use an **escape character.** In Python, the backslash \ is an escape character, and it forms an **escape sequence** when combined with the character that follows it. Here's a list of common escape sequences:

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| \n | Insert a newline character |
| \t | Insert a horizontal tab |
| \\ | Insert a backslash character |
| \" | Insert a double quote character |
| \' | Insert a single quote character |

Table 3: Escape sequences



print("Build \nFor \nWhat's \tNext.")

This yields:

Build For

What's Next.

A str can be as long as you need, but writing a long string on one line might reduce readability. You can use a backslash \ at the end of a line to split a long string across several lines without breaking it into multiple lines. For example:

###### printing a long line

print("You just finished your first semester.\ This semester you are taking four classes.\

During Spring 1 you will take four classes, as well.")

This yields:

You just finished your first semester.This semester you are taking four classes.During Spring 1 you will take four classes, as well.

As mentioned, a string is a sequence of characters (symbols). You can find the number of characters in a str

using Python's built-in lenO function. Here is an example:

###### finding length of a string

s1 = "Data"

s2 = "Year 2025"

print("length of s1:", len(s1)) print("length of s2:", len(s2))

yields

length of s1: 4 length of s2: 9



lenO is a built-in function that can return the number of characters in a string6

###### String Concatenation (Joining)

Attaching (joining) two strings using + operator to form a new string is called **string concatenation.**

6The lenO function can also be used to find the number of items in sequences such as list, tuple, or collections such as diet and

set. More on this later.

**concatenating two strings**

s1 = "2025 will"

s2 ="be great year" print(s1 + s2)

This yields:

2025 will be great year

You can concatenate as many strings as you need.

* + 1. str(), int 0, float(), **and** round() **Functions**

Besides being a data type, str O is a useful function that converts non-string values into a str.

**converting a number to a string**

num\_1 = 1.5

string\_1 = str(num\_1) print(type(string\_1))

This yields:

str

After converting a number to a str, you can no longer perform arithmetic operations on it unless you convert it back to a number using int O or float O. Let's learn more about these two new functions. int O is a function that converts a value to an integer, while float O converts a value to a floating-point number.

**converting a string to an integer**

str\_1 = "10"

var\_1 = int(str 1) type(var\_ 1)

This yields:

int

**converting a string to a float**

str\_2 = "3.14"

var\_2 = float(str 2) type(var\_2)

This yields:

float

By using int O and float O,you can convert strings back to numbers for performing calculations.

Now that we have learned about int O and float O function, let's look at round() function as well. The round() function is used to round a floating-point number to a specified number of decimal places. If no number of decimal places is provided, it rounds to the nearest whole number.

**Syntax:**

* round (number): Rounds number to the nearest integer.
* round(number, ndigi ts): Rounds number to ndigi ts decimal places.

**rounding to the nearest integer using** round() **function**

num\_1 = 4.67

rounded\_num\_1 =round(num\_1) print(rounded\_num\_1)

this yields:

5

In this example, round() rounds 4. 67 to the nearest whole number, which is 5.

**rounding to two decimal places using** round() **function**

num\_2 = 3.14159

rounded\_num\_2 =round(num\_2, 2) print(rounded\_num\_2)

this yields:

3.14

In this example, round() rounds 3 .14159 to 2 decimal places, resulting in 3 .14. str(), int(), and float(), and round() functions

str O function converts a value to a string.

int O function converts a value to an integer.

float O function converts a value to a floating-point number.

round O function rounds a floating-point number to the nearest integer or to a specified number of decimal places.

* + 1. input O **Function**

input O is a Python built-in function for getting input from the user. This function always returns a str.



input("What is your name?") # *This will ask the user to enter a name*

Running this line will prompt the user to enter their name and return it as a string.

**concatenating user input rather than calculations**

input1 = input("Enter first number") # *Asks the user to enter a number* input2 = input("Enter second number") # *Asks the user to enter another number* print(input1 + input2) # *Concatenates user inputs*

Since the input O function always returns a str, this will concatenate the two inputs as a single string **input1input2,** rather than performing any calculations. To work with numbers, you'll need to convert the inputs to the appropriate type, using int O or float O for numerical inputs. Here's an example:

**summing user input as** float

# *get user inputs and convert them to float* input1 float(input("Enter first number")) input2 = float(input("Enter second number"))

# *print sum of user inputs*

print(input1 + input2)

This will return the arithmetic sum of the two numbers.

* + 1. **Creating a String**

So far we have seen several different way of creating a string. Let's review these six methods of creating a string in Python here:

**Single Quotes**

# r•@+fii:l·iii·i'Mifil-iiifOiiiiiil"&

1 = 'Yes'

**Double Quotes**

# r•Mi:i+®Hii·i'ifaihiifi\Hiiiil"&

2 = "Yes"

**Triple Single Quotes (Useful for Multi-Line Strings)**

**using triple single quotes to create a multi-line string**

s3 = '' 'Yes python is fun'''

**Triple Double Quotes (Useful for Multi-Line Strings)**

**using triple double quotes to create a multi-line string**

s4 = """Yes python is fun"""

str O **Function**

# 'H'l"MNll-iii¥0¥Hiiil"h

5 = str(12)

**Concatenation**



s6 ="Carey"+ "Business"+ "School"

* + 1. **Comments**

In the examples above, we used comments to explain the code and make it easier to read. Python ignores anything after a #, treating it as a comment. Comments are helpful for understanding your code later, and they also help others (like your TA or professor) follow your work.

These things can improve code readability:

* + - * Comments
      * Spaces
      * Blank lines

Blank lines, spaces, and tabs are called **white space.** Python ignores most **white space,** but not all!7

* 1. bool
     1. bool **Basics**

Often in programming, we need to evaluate a statement. **Booleans** (or bool) represent one of two values: True or False. Python returns True if the statement is correct; otherwise, it returns False.

71ndentation using spaces is important in Python, which we will *cover* later.

###### checking boolean conditions

# *Assign prices to three cryptocurrencies and check a few conditions*

BitCoin, Ethereum, Ripple= 62\_000, 3\_000, 0.53 *#Using\_ for readability*

print(BitCoin > Ethereum) print(Ripple >= Ethereum)

print(Ripple != BitCoin)

print("Milo" 'Milo')

print("Milo" 'Milo ')

# *True if BitCoin > Ethereum, otherwise False*

# *True if Ripple>= Ethereum, otherwise False*

# *True if Ripple is not equal to BitCoin, otherwise False*

# *True if two strings are the same*

# *False due to the extra space*

This yields:

True False True True False

The == operator checks equality, which is different from the assignment operator =. Also, remember that white spaces inside quotation marks are part of the str, which is why the last statement is False!

###### Comparison Operators

True and False are Python keywords. A **condition** is a Boolean expression with a value of True or False. You can create conditions using the comparison operators in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Algebraic Operator** | **Python Operator** | **Sample Condition** | **Meaning** |
| > | > | X>Y | x is greater than y |
| < | < | X<Y | x is less than y |
| > | >= | **X** >= **y** | x is greater than or equal to y |
| < | <= | **X** <= **y** | xis less than or equal toy |
|  |  | **X** == **y** | x is equal to y |
| *i-* |  | **X** != **y** | x is not equal to y |

Table 4: Comparison operators

* + 1. **Boolean Operators** and, or, **and** not

Keywords and, or, and not combine simple conditions into more complex ones.

###### using boolean operator and

age= float(input("Enter your age as a number between 0 and 150")) serious\_medical\_history = input( "Do you have a serious medical history? y/n").lower()

result= (age>= 55) **and** (serious\_medical\_history

print(result)

"y")

This snippet asks the user for their age, converts it to a float, and asks if they have a serious medical history. The input is converted to lowercase using the lower O function. The condition checks if the age is at least 55 and if the user has a serious medical history. It returns True if both conditions are met, otherwise False, and then prints the result.

You can also use or and not to create different conditions. Here's an example:

###### using boolean operators or and not

age= float(input("Enter your age as a number between O and 150")) serious\_medical\_history = input("Do you have a serious medical history? y/n") .lower()

result= (age< 55) or (serious\_medical\_history == "n") print(not result)

The result will be True if the user is younger than 55 or does not have a serious medical history. The not

operator then inverts the result before printing it.8

lower() and upper() functions

The lower O function converts all uppercase letters in a string to lowercase, leaving other characters unchanged. Similarly, the upper O function does the opposite by converting all lowercase letters to upper­ case.

and, or, and not operators

The and operator returns True if both conditions are true. The or operator returns True if at least one condition is true.

The not operator inverts the result, returning True if the condition is false.

#### More on Strings

* + 1. **Membership Operators: in and** not in **keywords**

You can use the keyword in to check if a specific phrase or character is in a str. Similarly, the keyword not in

checks if a phrase or character is absent. These are Python's membership operators.

**using in and** not in **with strings**

s1 "Business Analytics" s2 "Isabella"

8Later on, we will see that Boolean operators are often used with conditional statements.

print("Analytics" in s1) print("bella" not in s2) print("Ness" in s1)

This yields:

True False False

in and not in operators

The in operator checks if a value exists within a sequence (like a string) and returns True if found. The not in operator does the opposite, returning True if the value is not present in the sequence.

###### Accessing Characters in a String

A str is a sequence of characters, and you can access each character using brackets [J . Each character has an integer index starting from 0.

###### accessing string characters by index

dog\_name = "Milo, The Golden!" first\_char = dog\_name[O] second char dog\_name[1] fourth\_char = dog\_name[3]

print("1st character of", dog\_name, "· II first char) print("2nd character of", dog\_name, "· II second\_char) print("4th character of", dog\_name, "· II fourth char)

This yields:

1st character of Milo, The Golden! M 2nd character of Milo, The Golden! i 4th character of Milo, The Golden! o

The last character is at index len(dog\_name)-1 9. Try this:

###### accessing the last character of a string

last\_char = dog\_name[len(dog\_name)-1]

print("Last character of", dog\_name, "is " last char)

This yields:

9Alternatively, you can access the last character with index of -1. More on this later!

Last character of Milo, The Golden! is!

* + 1. **Immutability of a** String

A str in Python is **immutable,** which means that once you create a string, you cannot change its individual characters. Any attempt to modify a character of a string will result in an error. This immutability helps ensure that strings remain consistent and predictable throughout your program.

For example, if you try to change a character in a string using assignment operator =, like this:

**Immutability of a** string

str\_1 = 'Care'

str\_1[1] = 'o' # *Try to replace 'a' with 'o'*

It will produce the following error:

TypeError: 'str' object does not support item assignment

The error indicates that you cannot modify characters within a string using indexing, as Python does not allow assignment to individual elements of an existing string.

###### Why is this Useful?

Immutability of strings means that when you manipulate strings, Python often creates new strings rather than modifying existing ones. This can make your code more predictable, as you can be sure that a given string won't change unexpectedly after it's created.

###### Creating a New String

If you want to modify a string, you need to create a new one. For example, to change the second character of

str\_1 to 'o', you would do this:

**Creating a new** string

str\_1 = 'Care'

str\_2 = str\_1[:1] + 'o' + str\_1[2:] print(f'Original string: {str\_1}') print(f'Modified string: {str\_2}')

This yields:

Original string: Care Modified string: Core

Here, str\_2 is a new string based on str\_1, but with the character at index 1 changed to 'o'. This approach keeps the original string unchanged, which is a key aspect of working with immutable types in Python.

* + 1. **Formatting Strings**

An £-string (short for formatted string) allows you to insert variables or expressions directly into a string by placing them inside curly braces {}. This makes string formatting cleaner and easier to read.

Often, you may need to display output in a specific format. As a first example, let's display the sum of two integer numbers on the screen.

**basic** string **formatting**

number1 = 6

number2 = 10

print('Sum of', number1, 'and', number2, 'is', number1 + number2, '.')

This yields:

Sum of 6 and 10 is 16.

While this works, it's easy to make mistakes with commas, quotes, or spaces. A more Pythonic way to format strings is by using £-strings. With £-strings, you can directly insert variables and expressions into the string.

Let's redo the above example using £-strings:

**formatting with f-strings**

number1 = 6

number2 = 10

print(f'Sum of {number1} and {number2} is {number1 + number2}.')

This yields:

Sum of 6 and 10 is 16.

With £-strings, you simply put variables or expressions inside curly braces {}, and Python automatically re­ places them with their values.

£-strings are very flexible. One common use is controlling the display precision of numbers.

**controlling precision with f-strings**

a = 1 / 3

b = f '{a: . 4f}'

print(b)

This yields:

0.3333

In this case, . 4f tells Python to format the number with 4 digits after the decimal point. You can change the number of decimal places by adjusting the x in . xf. The f specifies that the number should be displayed as

a float.

Here's another example that formats the value of pi to different levels of precision:

###### formatting pi with f-strings

pi= 3.14159265

formatted Pi 1 f'{pi: .2f}'

formatted Pi 2 = f'{pi:.0f}' formatted\_Pi\_3 = f'{pi:.11f}' print(formatted\_Pi\_1) print(formatted\_Pi\_2)

print(formatted\_Pi\_3)

# *two digits after the decimal*

# *zero digits after the decimal*

# *eleven digits after the decimal*

This yields: 3.14

3

3.14159265000

Notice that using £-strings allows us to display the value of pi with different levels of precision. However, this formatting does not change the original value of the pi variable; it merely controls how the value is displayed.

£-strings can also include function calls or expressions directly inside the curly braces. Let's see an example:



length of

is {len("Python")}.')

This yields:

The length of "Python" is 6.

In this example, we call the len() function inside the f-string to get the length of the string "Python".

You can also align or pad your text using £-strings. For example, to align text to the left or right within a specific width, you can use:

###### alignment with f-strings

cheese\_name = "mozzarella"

print(f' l{cheese\_name:<15}I ') # *Left-align within 15 characters*

print(f' l{cheese\_name:>15}I ') # *Right-align within 15 characters*

This yields:

Imozzarella

mozzarella I

In this example, < is used to left-align and > is used to right-align, with the specified width of 15 characters.

£-strings also offer other powerful capabilities, such as formatting percentages, which we will discuss further throughout the semester.

£-strings

The £-string allows you to insert variables or expressions directly inside a string by placing them within curly braces {}. This makes string formatting easier, more readable, and flexible.

1. **help O Function**

The help O function is a built-in function in Python that allows you to learn more about variables, functions, modules, and other components. It provides helpful documentation and explanations about how to use a specific function or module. You can use help O to get information about any variable, Python function, or module. Let's see an example where we ask for help on the print O function:

**t'Mi:IHMl+MHU**►

elp(print)

This will display information on how the print O function works, along with its arguments and examples.

If you are unsure about what a function does, the help() function is a quick way to understand its purpose and how to use it.

**Note for Jupyterlab users:** In Jupyterlab, you can use a shortcut by typing ? after any variable, function, or module name to get quick information. For example, run print? in a Jupyterlab cell to see a brief description of the print O function. You can use this with variables and modules too.

help() function

The help O function provides detailed information about variables, functions, and modules. It is a built-in way to access Python's documentation and understand how to use different components.

### Program Control

#### Introduction

Solving any computational problem requires executing a set of actions in a particular order. An **algorithm** is a list of instructions used for performing tasks. A cooking recipe found in any cookbook is an example of an algorithm. Any recipe has a list of instructions (actions) and the order of these actions.

In the context of programming, an algorithm is a set of procedures used for solving a problem. Any algorithm contains:

* The actions to be executed
* The order in which these actions are executed

A Python program is a list of actions performed by the computer in a particular order. Python uses three different forms of program control for specifying the order of instructions:

* **Sequential execution**
* **Selection statement**
* **Repetition statement**

Most programs use a combination of these program controls.

**5.1.1 Sequential Execution**

In sequential execution, as the name suggests, statements are performed in the order they are written. Here is an example:

**sequential execution of average**

Python executes these instructions sequentially to find the average of two numbers.

**5.2 Selection Statement**

A selection statement executes code based on a condition (an expression that evaluates to either True or False).

Here is an example:

**selection statement based on grade**

grade= float(input("Enter your grade")) if grade>= 70:

print(f"{grade} is a passing grade")

**else:**

print(f"{grade} is a failing grade")

print("We are done!")

X = 10

y = 20

temp= x + y average= temp/ 2

print(f"Average of {x} and {y} is {average}")

In programming, an algorithm is a step-by-step process for solving a problem or performing a task using a sequence of instructions.

Algorithm

This snippet does not perform instructions sequentially. Only one of the 3rd or 5th lines will be executed based on the condition. If grade >= 70 is True, Python skips the else block (line 5); and if grade < 70, Python executes the else block instead and skips line 3.

* + 1. **if Statement**

An if statement runs a set of actions if a condition is True and skips them if the condition is False.10

In the example below, the user inputs a grade, and the program prints "Pass" and "Great job!" if grade >=

70.

###### if statement with grade

grade= float(input('Enter a grade'))

if grade>= 70:

print ( 'Pass' )

print ( 'Great job! ' )

An if block is a group of statements that will only run if the condition is True.Every statement inside (below) the if block must be indented consistently. Python uses indentation (spaces or tabs) to define the structure of the code, so inconsistent indentation will cause an IndentationError.

* + 1. if . else **Statement**

The if . . . else statement performs one action if a condition is True, and a different action if the condition is False. The following snippet shows different messages based on the grade received from the user.

if else **statement with grade**

grade= float(input('Enter a grade'))

**if** grade>= 70:

print ( 'Pass' )

print ('Great job! ')

**else:**

print ('Failed')

print('Try harder next time')

**Concise Alternative:** When suites inside if and else are very short, you can put the whole if . . . else block in one line. Here's an example:

**concise** if . . else **statement**

grade= float(input('Enter a grade')) print('Pass') if grade>= 70 **else** print('Fail')

This snippet will take a grade from the user and print "Pass" if grade >= 70; otherwise, it will print "Fail".

10**A technical note:** The if condition can be any expression. Non-zero values are treated as True, while zero is treated as False.

Non-empty strings are also True, while empty strings are False.

* + 1. if . elif . else **Statement**

The if ... elif ... else statement runs only one of many possible actions depending on which condition is

True. The keyword elif is short for else if.

The following snippet accepts a floating-point input from the user and prints the corresponding grade.

if elif else **statement with grade**

grade= float(input("Enter your grade")) if grade>= 90:

print(f"grade of {grade} will get you an A") print("Great job")

**elif** grade>= 80:

print(f"grade of {grade} will get you a B")

**elif** grade>= 70:

print(f"grade of {grade} will get you a C") print("Let's talk.")

**else:**

print(f"grade of {grade} is not acceptable")

It is important to understand that only the action for the first True condition is executed, even if other conditions are also True. For example, if grade = 93, Python will run only the block for grade >= 90 even though 93 >= 80 and 93 >= 70 are also true.

The else statement in if . . . elif . . . else is optional and handles cases where none of the conditions are satisfied.

#### Repetition Statements

Repetition statements (sometimes called loops) are used to repeat the same code multiple times in succession. Python has two types of loops:

* while **statement:** Repeats an action (or a group of actions) as long as a condition remains True. This is known as a condition-controlled loop.
* for **statement:** Repeats an action (or a group of actions) for a specific number of times. This is known as a count-controlled loop.

For a better understanding of the difference of these two loops, let's look at their basic syntaxes (structures). The basic syntax for a for loop is:

* + 1. while **Loop**

A while loop repeats a block of code as long as a condition is True. The basic syntax for a while loop is:

while condition:

statement(s) # These statement(s) must be indented

Here's how a while loop works:

* + - * Test the condition before the loop starts.
      * If the condition is True, repeat the block again.
      * If the condition is False, exit the loop.

The following snippet keeps track of the user's total score. It will ask for scores and keep adding them to the total until the user decides to stop.

**adding scores using** while **loop**

total\_score = 0 keep\_going = "yes"

**while** keep\_going == "yes": # *Loop as Long as the user wants to continue* score= int(input("Enter your score: ")) # *ask for a score* total\_score += score # *add the score to the total*

print(f"Total score so far: {total\_score}") # *print the updated total* keep\_going = input("Add another score? yes/no").lower() # *ask if they want to* '-+ *continue*

print(f"Final total score: {total\_score}")

**Be aware of infinite loops:** The condition inside the while loop must eventually become False; otherwise, the loop will continue infinitely, known as an **infinite loop.** Here is a simple example of an infinite loop:

**infinite loop counting up**

|  |  |
| --- | --- |
| i = 0 |  |
| indicator= True |  |
| while indicator True: | # *This condition will always stay True* |
| print(f"i = {i}") |  |
| i += 1 | # *Increment* i *by 1 each time* |

This loop will keep counting up forever, or until the computer crashes, because the condition indicator == True will never become False.

Now, here's a modified version with an exit condition:

while **loop with exit condition**

i = 0

indicator= True

while indicator True: print(f"i = {i}")

i += 1

if i == 10:

indicator= False

In this improved version, the loop will stop when i == 10, because the indicator variable is set to False,

breaking the loop condition.

while Loop

**5.3.2 for Loop**

A for loop repeats a block of statements for a specific number of times. The basic syntax for a for loop is:

for variable in some collection:

statement(s) # These statements must be indented

The for statement executes its block of statements for each item in a collection of items.

Let's see a few examples.

for **loop iterating over characters in a string**

for item in "America": print(item)

A while loop repeats a block of code as long as a condition remains True. It stops when the condition becomes False.

This yields:

A

m

e

r

i

C

a

**Reminder:** A string is a sequence of characters. We can loop through these characters and access them individually.

for **loop with a** list

for itern in ["Good Dog", "Naughty Cat", "Loyal Dog"] : print(item)

This snippet will go through three different phrases "Good Dog", "Naughty Cat", "Loyal Dog" and print them individually yielding

11We will learn elaborately about list data structure later

Good Dog Naughty Cat Loyal Dog

for **loop with** range

2

3

4

The range O function generates a sequence of numbers. In this example, range(2, 5) produces numbers starting from 2 (inclusive) and ending before 5 (exclusive), so it generates 2, 3, and 4.

for Loop

**Side note: A Cool Property of the** print O **Function**

You can use the optional end argument in the print O function to control how the output is displayed:

for **loop with custom end**

for item in "America": print(item, end=" ")

A for loop repeats a block of code for each item in a sequence, running the loop until all items are processed.

for i in range(2, 5): # *range starts at 2 (incLusive) and stops at 5 (excLusive)*

print(i)

This yields:

A m e r i c a

Instead of printing each character on a new line, the end argument controls what is printed at the end of each output. In this case, instead of moving to a new line after each character, it prints each character with three spaces, determined by end argument, in between. This helps customize the output format.

I've noticed more frequently that people use underscores as identifiers. The underscore character \_ is valid for Python identifiers. Try:

# t'Hi:1-l'i:l-iiMMifHi·IU!ii►

or\_ in range(5):

print(\_)

It's worth noting that the underscore character \_ is frequently used as a valid identifier in Python. In some

cases, it is used when the value of a variable is not important. For example:

# t'Hi:1-l'i:l-iiHMifHi·IU!ii►

or\_ in range(3):

print(\_)

This yields:

0

1

2

In this example, the underscore \_ is used as a placeholder for the loop variable since we don't need to reference the actual value inside the loop. This is a common practice when the variable's value isn't needed but we still need to run the loop a certain number of times. For more on the underscore's role in Python, check out this reference on DataCamp: https://[www.datacamp.com/community/tutorials/](http://www.datacamp.com/community/tutorials/) role-underscore-python#IV.

* 1. range O **Function**

The range O function is a built-in Python function that generates a sequence of numbers. It simplifies count­ controlled for loops. The function can accept one, two, or three arguments:

* **With one argument:** range(a) produces a sequence from O up to, but not including, a. For example:

# tfHAAl'i:id!M:iflil;■'i:iHi'i::M:i►

or i in range(3): print(i)

This yields:

0

1

2

* **With two arguments:** range(a, b) produces a sequence from a up to, but not including, b. For example:

# tfHAAl'i:id!M:IWiiEf:Hii·i'i::M:i♦

or i in range(2, 5):

print(i)

This yields:

2

3

4

* **With three arguments:**range(a, b, s) generates a sequence from a up to, but not including, b, incre­ menting by s. If a < b, s must be positive. If a > b, s must be negative. For example:

**tfHHA!'i:MiM:IWiiii:iHii·i'i::i&:ia**

or i in range(2, 11, 3): print(i)

This yields:

2

5

8

**tfHHA!'i:MiM:i·iM::i&:iii:i·ht**

or i in range(20, 5, -9): print(i)

This yields:

20

11

range() Function

The range() function generates a sequence of numbers, often used to control the number of iterations in a for loop.

#### Concept of an lterable

We discussed earlier that a for loop is used for iterating over a sequence. The sequence to the right of the in keyword of the for statement must be an **iterable.** An iterable is something12 that the for loop can go through, one item at a time, until there are no more items left. The most common iterables in Python are:

* **A** string
* **A** range
* **A** list
* **A** tuple
* **A** dictionary
* **A** set

12obj ect, to be more accurate

We will gradually learn about each of these data structures in detail.

#### Nested Loop

A **nested loop** is a loop inside another loop. The "inner loop" will be executed one time for each iteration of the "outer loop."

###### nested loop example

adjectives= ['beautiful', 'fun', 'safe'] metropolitan= ['San Diego', 'New York City'] for adj in adjectives:

for city in metropolitan: print(f"{city} is {adj}")

print() # *print an empty Line*

This yields:

San Diego is beautiful New York City is beautiful

San Diego is fun New York City is fun

San Diego is safe New York City is safe

In the outer loop, adj takes the value 'beautiful'; then the inner loop goes through all the items in metropolitan and performs print(f"{city} is {adj}"). Once the inner loop is done, the outer loop prints an empty line using print O. This finishes the first iteration of the outer loop. adj then takes the value 'fun' in the outer loop, and the inner loop starts over again. The repetition continues until there is no item left in the outer loop iterable.

**Note:** You can put any type of loop inside any other type of loop. For example, a for loop can be inside a while

loop or vice versa.

Nested Loops

A nested loop is a loop inside another loop. The inner loop runs completely for each iteration of the outer loop.

* 1. **Optional:** Break **and** Continue

Python allows:

* Stopping the whole loop before going through all the items in the iterable (using break).
* Stopping the current iteration of the loop and continuing with the next (using continue).
* A for loop to have an else block as well.

These are not commonly used features in Python, but interested students can learn more from the official documentation at https: //docs. python. org/3/tutorial/controlflow. html.

1. **Concept of an** object **and a** class

Python is an object-oriented language, meaning everything in Python is treated as an **object.** But what is an object? Simply put, an object is a value stored in the computer's memory. Every object contains some data and has the ability to perform actions (called methods).

For example, in the statement name = "Milo", the value "Milo" is an object. The variable name points to (refers to) that object.

#### Methods

A method is a special function that belongs to an object and performs an action. You can use a method by attaching it to an object with a dot (. ).

For example, the string "Milo" has a method called upper O, which converts all letters to uppercase.

**string method** upper O

name= "Milo"

print(name.upper()) # *Call the method 'upper()' on the string 'Milo'*

This yields:

MILO

#### Types and Classes

Every object in Python belongs to a **class.** A class is like a template that defines the behavior (methods) of objects of that type.

For example:

* Strings like "Milo" belong to the str class.
* Numbers like 100 belong to the int class.
* Decimal numbers like 99.8 belong to the float class.

You can use Python's type O function to check the class of an object:

**checking object type**

name= "Milo" print(type(name)) # *str class*

age= 5

print(type(age)) # *int class*

weight= 22.5

print(type(weight)) # *float class*

This yields:

<class 'str'>

<class 'int'>

<class 'float'>

#### Methods and Classes

Objects that belong to the same class respond to the same methods. For example, both 'Spring' and 'Fall'

are str objects, so they both respond to the upper() method.

**using** upper O **on two strings**

season1 = "Spring" season2 = "Fall"

print(season1.upper()) # *Both are strings, so upper() works on both*

print(season2.upper())

This yields:

SPRING FALL

Each data type in Python is associated with a class. For example:

* x = 10 creates an object from the int class.
* y = 99.8 creates an object from the float class.

By knowing the class of an object, you can understand which methods you can use with it!

class, object, and method

**A** class is a template for creating objects. An object is created from a class, and it contains data and functions (methods). A method is a function inside a class that defines what an object can do.

#### Creating and Deleting Objects

In Python, objects are created automatically whenever you assign a value to a variable. This variable then becomes a reference to the object.

Here's a simple example:



name= "Milo" *#Anew string object is created*

age= 9 *#Anew integer object is created*

In this example:

* name refers to a string object "Milo".
* age refers to an integer object 9.

These objects remain in memory as long as they are referenced by at least one variable. However, Python allows you to delete references to objects using the del keyword. This doesn't delete the object immediately but rather removes the variable's reference to it.

Here's an example of deleting an object reference:

**deleting objects with** del

adj= "Difficult"

**del** adj # *The reference to the object "Difficult" is removed*

After using del, the variable adj no longer refers to the object. If you try to access adj after deletion, you will get an error.

NameError: name 'adj' is not defined

Python automatically deletes objects that are no longer referenced by any variable. This process is known as **garbage collection.** While you can use del to explicitly remove a reference, Python will take care of deleting the object when it's no longer needed.

del Keyword

The del keyword is used to delete a variable or object reference. Once deleted, the variable can no longer be used unless it is redefined.

##### str More Elaborate Methods

So far, we have seen the lower O and upper O methods for string objects. This section introduces additional methods, as listed in the following table.

|  |  |  |
| --- | --- | --- |
| **Method** | **Sample Syntax** | **Description** |
| replace() | str.replace(old, new) | Replaces occurrences of old with new in a string |
| split() | str.split(separator) | Splits a string into a list of substrings based on  the separator |
| join() | separator.join(iterable) | Concatenates elements of iterable with the  specified separator |
| strip() | str.strip() | Removes leading and trailing whitespace (or  specified characters) from a string |
| startswith() | str.startswith(prefix) | Checks if the string starts with the specified  prefix |
| endswith() | str.endswith(suffix) | Checks if the string ends with the specified  suffix |

Table 5: String Methods, Syntax, and Descriptions

* + 1. replace() **Method**

The replace O method replaces occurrences of an old value with a new one in a string.

replace O **usage**

lie= "Java is a fun language"

truth= lie.replace('Java', 'Python') # *Replace 'Java' with 'Python'*

print(truth)

This yields:

Python is a fun language

* + 1. split O **Method**

The split O method splits a string into smaller pieces based on a separator.



course name= "Statistical\_Analysis Python Dperations\_Management Simulation" print(course\_name.split()) # *Split by whitespace*

This yields:

['Statistical\_Analysis', 'Python', '0perations\_Management', 'Simulation']

* + 1. join() **Method**

The join() method concatenates elements of an iterable with a separator.



words 1 = ['Are', 'you' , 'ready' , 'for' , 'a', 'fun' , 'semester' , '?'] str\_1 = ' ' # *a space*

print(str\_1.join(words\_1)) # *Join words with a space*

This yields:

Are you ready for a fun semester?

* + 1. strip() **Method**

The strip O method removes unnecessary characters from the beginning and end of a string.

strip O **usage**

str 1 = '

We are almost done with this handout

print(str\_1.strip()) # *Strip white spaces*

This yields:

We are almost done with this handout

* + 1. startswithO **Method**

The startswith O method checks whether a string starts with a specified value.

startswith O **usage**

dog\_name = "Milo, The Naughty Golden Retriever!" print(dog\_name.startswith('Milo')) print(dog\_name.startswith('Golden')) print(dog\_name.startswith('Th', 6)) # *Start search at index 6*

This yields:

True False True

* + 1. endswi th() **Method**

The endswi th O method checks if a string ends with a specified value.

endswi th() **usage**

statement\_1 = "Why do I have homework during my break?!" print(statement\_1.endswith('break?! ')) print(statement\_1.endswith(' ! '))

This yields:

True True

str Methods

* lower O converts all uppercase characters in a string to lowercase.
* upper O converts all lowercase characters in a string to uppercase.
* replace O replaces all occurrences of a specified substring with another substring.
* split O splits a string into a list of substrings based on a specified separator.
* join() concatenates elements of an iterable with a specified separator, returning a single string.
* strip O removes leading and trailing whitespace or specified characters from a string.
* startswi th() checks if a string begins with a specified prefix.
* endswi th O checks if a string ends with a specified suffix.

### Python Standard Library

A **library** is a collection of reusable code that provides specific tools and capabilities. A **module** is a part of a library that can be used in your programs. Usually, a library consists of several modules.

Python comes with a **standard library** that includes many built-in modules, offering an extensive set of capa­ bilities. Additionally, Python supports external libraries like numpy and pandas for more specialized tasks.13 The following table summarizes some of the commonly used **modules** in Python's standard library:

These modules form a foundation for many types of tasks in Python, from file handling and data manipulation to system operations and statistical analysis. To use a module from the **standard library,** you must import it first.

A complete list of these capabilities is available at **Python Standard Library Documentation** available at

https: //docs. python. org/3/library/. Here are a few examples:

13A **module** is typically a single file that contains Python code, such as functions, classes, and variables, that can be imported and used in your programs. Libraries are made up of one or more modules.

|  |  |
| --- | --- |
| **Module** | **Application** |
| csv | Processing comma-separated value files |
| datetime,time | Date and time manipulation |
| decimal | Accurate calculations |
| math | Common math constants and operations |
| OS | Interactions with the operating system |
| sys | System-specific parameters and functions |
| timeit | Performance analysis |
| random | Random number generation |
| statistics | Statistical functions like mean, median, and variance |
| string | String processing |
| json | Working with JSON data |
| re | Regular expression pattern matching |
| itertools | Tools for working with iterators |

Table 6: Python standard library modules

**Example from** math **Module**

**using** math **module**

import math print(math.pi) print(math.factorial(5))

# *import math module*

# *print pi*

# *returns 5!* = *120*

This yields:

3.141592653589793

120

**Example from** random **Module using** random **module**

import random

# *import random module*

print(random.choice(['Python', 'Java', 'Julia'])) # *choose one randomly*

print(random.random()) # *a random number between* O *and 1*

This yields (your answers will be different):

Julia 0.20713890285322634

**Example from** statistics **Module using** statistics **module**

**import statistics** # *import statistics module*

print(statistics.variance([20, 12, 15, 11])) # *find sample variance*

print(statistics.mean([20, 12, 15, 11])) # *find sample mean*

This yields:

16.333333333333332

14.5

**Example from** datetime **Module using** datetime **module**

from datetime import date ## *import date class from datetime module*

now= date.today() ## *define now*

print(now)

This yields:

2024-10-06

The Python Standard Library is a collection of modules that provide useful functions and tools to perform a wide range of tasks, from file handling to mathematical operations, without the need for external libraries.

Python Standard Library

### Working with Precision: Understanding Decimal in Python

In this section, we will explore Python's decimal module, which allows us to perform highly accurate calculations, especially when precision is crucial.

#### Motivation

In everyday life, we use base-10 (decimal) numbers, which means we count in powers of ten (1, 10, 100, etc.). For example, the number 123 represents 1 x 102 + 2 x 101 + 3 x 10°. Computers, however, use base-2 (binary), where everything is represented as Os and 1s. This fundamental difference between how we naturally represent

numbers and how computers store them can result in small inaccuracies when performing arithmetic operations with decimal values.

Let's see an example performing basic arithmetic calculations where results are displayed with 25 decimal places:

###### Floating-Point Arithmetic Example

print(f'1/2 with 25 decimal point precision is {1/2:.25£}') print(f'1/3 with 25 decimal point precision is {1/3:.25£}') print(f'1/10 with 25 decimal point precision is {1/10:.25£}') print(f'0.1 + 0.2 with 25 decimal point precision is {0.1+0.2:.25f}')

This produces the following results:

1/2 with 25 decimal point precision is 0.5000000000000000000000000 1/3 with 25 decimal point precision is 0.3333333333333333148296163 1/10 with 25 decimal point precision is 0.1000000000000000055511151

0.1 + 0.2 with 25 decimal point precision is 0.3000000000000000444089210

Notice that while 1/2 is displayed accurately, the results for 1/3, 2/9, and o. 1 + o. 2 are slightly different from what we would expect. For example, 1/3 should ideally be o. 3333333333333333333333333, but instead, it displays as o.3333333333333333148296163-a small yet noticeable discrepancy.

In Python, numbers with decimal points (like 0.1 or 2.75) are stored using a system called **floating-point rep­ resentation**14. While this format is efficient for representing very large or very small numbers, it cannot store all decimal numbers with perfect accuracy. As a result, arithmetic operations with floating-point numbers can sometimes produce small rounding errors.15 These inaccuracies are problematic when precision is critical, such as in financial calculations, scientific measurements, or any scenario where even small errors are unacceptable.

To address this, Python offers a special data type called Decimal from the decimal module. Unlike floating­ point numbers, Decimal stores numbers as **base-10** fractions, similar to how humans naturally work with deci­ mals, ensuring more accurate and precise results. Since every decimal number can be represented precisely in base-10, calculations like 0.1 + 0.2 will result in exactly 0.3 without any extra rounding errors.

* 1. **Using** Decimal

To use the Decimal type, the first step is to import the decimal module and create Decimal object. These objects allow for precise arithmetic. Next, it is important to pass the number as a string rather than a floating-point number, since floating-point numbers may already contain precision errors.

Here's an example of using Decimal for accurate arithmetic:

**Using** Decimal **for Precise Arithmetic**

from decimal import Decimal

result= Decimal('0.1') + Decimal('0.2') print(result)

In the example above, the numbers o. 1 and o.2 are passed as strings to avoid any potential floating-point

14For more technical details, refer to https: //en. wikipedia. org/wiki/Double-precision\_floating-point\_format

15Think of it like trying to represent the fraction ½ in decimal format-it results in an endless repeating decimal (0.3333...), and any attempt to store it exactly will always involve some rounding.

rounding issues. The result is an exact representation of o. 3, as shown below:

0.3

* 1. **Operations Using** Decimal

Like regular numbers, you can perform various arithmetic operations using Decimal.

Here's an example of using basic arithmetic with Decimal:

**Arithmetic with** Decimal

from decimal import Decimal

# *Arithmetic operations*

a= Decimal('2.5')

b = Decimal('1.3') addition= a+ b subtraction= a - b multiplication= a\* b division= a/ b

print(f'Addition: {addition}') print(f'Subtraction: {subtraction}') print(f'Multiplication: {multiplication}') print(f'Division: {division}')

This yields:

Addition: 3.8

Subtraction: 1.2

Multiplication: 3.25

Division: 1.9230769230769231

decimal Module and Decimal Type

The decimal module provides the Decimal data type, allowing for precise base-10 arithmetic. It is espe­ cially useful in applications where accuracy is critical, such as financial and scientific calculations.

#### Why Not Use Decimal for All Calculations?

You may wonder why Python doesn't do all the calculations in base-10 using Decimal. Here is why: The main drawback of using the Decimal data type in Python is **performance.** Arithmetic operations with Decimal are slower compared to floating-point operations. This is because Decimal provides more precision and accuracy by performing base-10 arithmetic, which requires additional processing to avoid the rounding errors associated with floating-point arithmetic. While this trade-off is acceptable in applications where precision is critical, such as

financial calculations, it can be a disadvantage in scenarios where performance is more important.

Next Topic Preview

In the next handout, we will explore Python's collection data types, including list, tuple, dictionary, and set. These collections allow us to store multiple values together, making it easier to manage and manipulate groups of related data. Understanding these data types is crucial for more advanced data handling and efficient programming in Python.

## Exercise Questions

1. Write a for loop to print the numbers from 1 to 5. Use the range() function.
2. Write a while loop that prints the numbers from **1** to 5.
3. Write a for loop to print each character in the string "Hello DC".
4. Accept a number from the user using input O function. Then, use an if block to check if the number is positive or negative. Print "Positive" if the number is greater than zero; otherwise, print "Not Positive".
5. Use a for loop to calculate the sum of all numbers from 1 to 10.
6. Generate 5 random integers between 1 and 10 using random. randint () inside a for loop. Import the

random module first.

1. Write a while loop that keeps generating a random number between 0 and 1 until the number is greater than 0.8. Print each generated number.
2. Take a user's name using input O function. Use an if ... elif ... else block to check if a given string starts with "A", "B", or some other letter which in this case you should mention that the first letter of the name. Print a different message for each case.
3. Write a for loop to print the squares of numbers from 1 to 10. Use the range O function.
4. Write a program that takes a number as input and prints "Even" if it is even, and "Odd" if it is odd. Use the

% (modulus) operator and an if ... else block.

1. Write a for loop to print every third character of the string "Python Programming is fun! " starting from the first character.
2. Write a program that generates a random floating-point number between 0 and 10, rounds it to 2 decimal places, and prints the result. Use the random module and round() function.
3. Write a program that asks the user to enter a word, and then checks if the word contains the letter "e".

Print "Contains 'e'" if it does, otherwise print "Does not contain 'e' ". Use the in keyword.

1. Write a while loop that asks the user for a number and calculates the cumulative sum until the user enters

o. Print the cumulative sum at the end.

1. Write a program that asks the user for a temperature in Celsius and converts it to Fahrenheit using the formula F = C \* 9/5 + 32. Print the result using an £-string.
2. What is the difference between = and ==?
3. Write a code that accepts 4 numbers from the user and prints *a// equal* if they all are equal numbers and prints *not equal* if not all numbers are the same.
4. Import random module and generate 16 uniform random numbers between 0 and 1. You need to use

random. random() function inside a for loop.

1. Import random module and generate 10,000 uniform random numbers between 0 and **1.** You need to use

random. random() function inside a for loop.

* 1. Find *sum* of these numbers.
  2. Find *average* of these numbers. You need to find sum and divide it by 10,000.
  3. Find *minimum* of these numbers? Did you get an exact Oas the minimum?
  4. Find *maximum* of these numbers? Did you get the exact 1 as the maximum?

1. Inside a for loop generate 5000 uniform random numbers between 0 and 1. Use an if block to print those numbers that are bigger than 0.5. How many numbers did you get?
2. There are several ways to generate standard normal random numbers. One method is to start with two uniform[0, 1] random numbers u\_1 and u\_2 and apply the following functions to get two standard normal random numbers:

Z1 = J-2 ln U1 cos(21rU2) Z2 = J-2 ln U1 sin(21rU2)

* 1. Use random and math modules to generate two standard normal random numbers using the above formulas. You need to explore functions available to you in math module.
  2. Using a for loop, generate 20 standard normal random numbers using the above method.
  3. Using a for loop, generate 2000 standard normal random numbers using the above method and keep only the positive numbers. How many positive numbers did you get?
  4. Inside a while loop generate enough random numbers to get the first number bigger than 3. How many numbers did you have to generate to get to the first number bigger than 3?
  5. Inside a while loop generate enough random numbers to get the 5 numbers that either are less than

-2.5 or more than 2.5. How many numbers did you have to generate to get this result?

1. We learned that True and False represent boolean values in Python. We know that computers store everything in memory using a **binary** format, which means only Os and 1s. In this binary system, True is stored as 1, and False is stored as o. Run this code to understand this behavior:

**numerical values of** True **and** False

print(f'True +True= **{True+ True}')** print(f'True +False= **{True+ False}')** print(f'False +False= **{False+ False}')** print(f'True \* 5 **={True\*** 5}') print(f'True >False= **{True> False}')**

What do you observe? How do you think Python treats True and False when used in calculations, given that they are stored as 1 and o in memory?

1. It is very tempting to use max, min, sum, type, and id as variable names. This question is designed to show you why this is a terrible idea. Running the following code

###### trouble using built-in functions as variables

num\_1 = 5

num\_2 = 6

sum= num 1 + num\_2 print(sum)

print (sum(1, 3))

will lead to TypeError: 'int' object is not callable. Why do you think that is the case?

1. We learned that we can compare numerical values using comparison operators such as=,<,>,<=,>=. This assignment will help you extend that to str. Run this code and try to understand how Python compares two strings

**string comparisons**

s\_1 = "Adams" s\_2 = "James" s\_3 = "Zara"

print(f'{s\_1} < {s\_2} results in {s\_1<s\_2}') print(f'{s\_3} < {s\_2} results in {s\_3>s\_2}')

How do you think Python compares two strings?

1. Using the Decimal data type, multiply o.1 by 3 and display the exact result.
2. You are working on a financial application that requires precise interest rate calculations. Using the Deci mal type, calculate the compounded value of an investment of $1,000 at an annual interest rate of 4.5%, compounded monthly, for 5 years. Use the formula:

where:

* *P* = 1000 (the principal)
* *r* = 0.045 (annual interest rate)
* *n* = 12 (compounding periods per year)
* *t* = 5 (time in years)

*A=P* (

1+;;*r*:*)nt*

Ensure precise calculation without any rounding errors.

## Exercise Solutions

Solutions to these problems can be found on the following GitHub page:

https://tinyurl.com/3wekv7mt

You can also access the same link using the QR code below:



1. **References**

**References and Resources**

The following references and resources were used in the preparation of these materials:

1. Official Python website at https: / /[www.python.org/.](http://www.python.org/)
2. *Introduction to Computation and Programming Using Python,* John Guttag, The MIT Press, 2nd edition, 2016.
3. *Python for Data Science Handbook: Essential Tools for Working with Data,* Jake VanderPlas, O'Reilly Media, 1st edition, 2016.
4. *Data Mining for Business Analytics,* Galit Shmueli, Peter C. Bruce, Peter Gedeck, Nitin R. Patel, Wiley, 2020.
5. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and /Python,* Wes McKinney, O'Reilly Media, 2nd edition, 2017.
6. *Introduction to Python for Computer Science and Data Science,* Paul J. Deitel, Harvey Deitel, Pear­ son, 1st edition, 2019.
7. *Data Visualization in Python with Pandas and Matplotlib,* David Landup, Independently published, 2021.
8. *Introduction to Programming Using Java,* available at http:/ /math. hws. edu/ javanotes/.
9. *Python for Programmers with Introductory Al Case Studies,* Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019.
10. *Effective Pandas: Patterns for Data Manipulation (Treading on Python),* Matt Harrison, Independently published, 2021.
11. Python tutorials at https: / /betterprogramming. pub/.
12. Python learning platform at https: / /[www.learnpython.org/.](http://www.learnpython.org/)
13. Python resources at https: / /real python. com/.
14. Python courses and tutorials at https: / /[www.datacamp.com/.](http://www.datacamp.com/)