Intro to Python

Strings

In Python, the `str` (string) data type represents a sequence of characters enclosed within single quotes (' ') or double quotes (" "). Strings in Python are immutable, which means they cannot be changed after they are created. Here are some key points about strings in Python:

Make a directory to run your scripts from so you can run them in a virtual environment.

When making a python file. start them with `#!/bin/python3` to call out the directory so we can simply `./scriptname.py`, if not, we will need to `python3 script.py`. Either way works.

Hello World!

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

This will print out "Hello World!" into our terminal.
You can use single quotes(') OR double quotes ("")

Notes/Comments

After certain sections, it's good practice to add comments with a poung sign (#) like:

```
print ("Hello World!") # Print String "Hello world!"
```

Multi Line Commenting

```
print ("""This string will run
on multiple lines""") # Triple Quote for multiple lines.
```

Concatenating

```
print ("This is a "+"concatenated string") # This will combine the 2 string
with the plus (+) symbol.
```

New Line/Line break

```
print ("Hello World!")
print ("\n") # This will make a new line
print ("This is on a new line")
```

Final Strings Notes

```
#!/bin/python3

# STRINGS
print ("Hello World") # Print String "Hello World!"
print ("""This string will run
on multiple lines""") # Triple Quote for multiple lines.
print ("This is a "+"concatenated string") # This will combine the 2 string
with the plus (+) symbol.
print ("\n") # This will make a new line
print ("This is on a new line")
```

Notes from course:

- Creation:
  Example:

```
my_string = 'Hello, World!' or my_string = "Hello, World!"
```

- Accessing Characters:
  You can access individual characters within a string using indexing, starting from 0.
  Example:

```
print(my_string[0]) would output 'H'.
```

- String Concatenation:
  You can concatenate (join) two or more strings using the + operator.
  Example:

```
greeting = 'Hello' + ' ' + 'World!' would result in 'Hello World!'.
```

- String Length:
  The len() function can be used to determine the length (number of characters) of a string.
  Example:

```
print(len(my_string)) would output the length of the string.
```

- String Slicing:
  You can extract a substring from a string using slicing, specifying the start and end indices.
  Example:

```
substring = my_string[7:12] would extract the substring 'World'.
```

- String Methods:
  Python provides various built-in methods to manipulate and transform strings. Examples include upper(), lower(), strip(), split(), replace(), and more.
  Example:

```
print(my_string.upper()) would output 'HELLO, WORLD!'.
```

Math

In Python, the math module is a built-in module that provides various mathematical functions and constants. It allows you to perform advanced mathematical operations in your Python programs. To use the math module, you need to import it first using the import statement.

Math Notes

```
#!/bin/python3

# MATH
print (50+50) # addition
print (50-2) # subtraction
print (50*50) # multiplication
print (50 / 2) # division
print (50 +50 - 2 * 50 / 2) # PEMDAS
print (50 ** 2) # exponents
print (50 % 6) # divison - "modulo" - Takes what it left over and shows the remainder
print (50 / 6) # division - float" - will show output with remainder
print (50 // 6) # division - with no remainder
```

Notes from course

Math Functions in the math Module:

- `math.sqrt(x)`: Calculates the square root of x.
- `math.pow(x, y)`: Raises x to the power of y.
- `math.exp(x)`: Calculates the exponential value of x (e^x).
- `math.log(x)`: Calculates the natural logarithm of x (base e).
- `math.log10(x)`: Calculates the logarithm of x to base 10.
- `math.sin(x), math.cos(x), math.tan(x)`: Calculate the sine, cosine, and tangent of x, respectively (where x is in radians).
- `math.degrees(x)`: Converts x from radians to degrees.
- `math.radians(x)`: Converts x from degrees to radians.

Math Operators:

- Addition (`+`): Adds two numbers.
- Subtraction (`-`): Subtracts one number from another.
- Multiplication (`*`): Multiplies two numbers.
- Division (`/`): Divides one number by another.

- Integer Division (`//`): Performs division and returns the quotient as an integer (rounds down).
- Modulo (`%`): Returns the remainder of division.
- Exponentiation (`**`): Raises a number to a power.

Example usage:

```
import math
# Using math functions
print(math.sqrt(25)) # Output: 5.0
print(math.pow(2, 3)) # Output: 8.0
print(math.sin(math.pi/2)) # Output: 1.0

# Using math operators
x = 10 y = 3 print(x + y) # Output: 13
print(x / y) # Output: 3.3333333333333335
print(x // y) # Output: 3
print(x % y) # Output: 1
print(x ** y) # Output: 1000
```

Variables and Methods

```
#!/bin/python3

# Variables and Mothods
print ("This section is about variables and methods")
print ("##########################################")
quote = "This is a quote." # Storing this string inside the variable called
"quote"
print (quote) # Will print what the variable "quote" is.

print (quote.upper()) # Prints in all caps/uppercase
print (quote.lower()) # Prints in all lowercase
print (quote.title()) # Title case - Caps on first letter in a word
print (len(quote)) # counts each character, including spaces

name = "th4ntis" # string
age = 32 # int or integer
years = 4.7 # float - has a decimal

print(int(age))
print (int(30.1)) # Shows just left side of decimal
print ("My name is " +name + " and I am " + str(age) + " years old.") #
concatenate strings
```

```
age += 1 # varible - adds 1 to the current variable of age
print (age) # Note variables can change

birthday = 1 # variable
age += birthday #
print (age)
```

Notes from course

In Python, variables and methods are fundamental concepts used in programming. Here's an explanation of each:

Variables:

A variable is a named storage location used to store data or values in a program. It acts as a placeholder for data that can be accessed, modified, or used in calculations throughout the program. Variables in Python are dynamically typed, meaning their data type can change during program execution. Here's an example of variable usage in Python:

```
# Variable assignment
x = 10
name = "John"
is_true = True


# Variable usage
y = x + 5
print("Hello, " + name)
if is_true:
    print("The condition is true")
```

In the example above, `x`, `name`, and `is_true` are variables assigned with different data types (integer, string, and boolean, respectively). They are used in calculations and print statements to perform operations and display values.

Methods:
A method is a block of reusable code that performs a specific task or action. Methods are associated with objects or classes and are called upon to perform certain operations. In Python, methods are commonly referred to as functions. Built-in functions and user-defined functions both fall under the category of methods. Here's an example:

```
# Built-in method example
numbers = [1, 2, 3, 4, 5]
length = len(numbers)
```

```
print("Length:", length)


# User-defined method example
def greet(name):
    print("Hello, " + name)


greet("Alice")
```

In the example above, `len()` is a built-in method that calculates the length of a list (`numbers` in this case). The user-defined method `greet()` takes a parameter `name` and prints a greeting message. It is called with the argument "Alice" to print "Hello, Alice" to the console.

Methods can have return values, perform actions, accept parameters, and more, depending on their purpose and design.

Variables and methods are essential components in Python programming. Variables store data, while methods encapsulate reusable blocks of code for specific tasks. Understanding their usage and relationship is crucial for building functional and efficient programs.

Funtions

In Python, a function is a reusable block of code that performs a specific task. Functions allow you to organize code into logical and modular units, making your code more readable, maintainable, and reusable.

```
#!/bin/python3
# FUNCTIONS
print ("This section is about Functions:")
print ("################################")
def who_am_i(): # This is a function without parameters
    name = "th4ntis" # This is a local variable
    age = 32 # local variable
    print ("My name is " +name + " and I am " + str(age) + " years old.")
    # Any variables stored in here, is only stored within the function, in
this case, the "who_am_i" function

who_am_i() # runs the "who_am_i" function we made above

def add_one_hundred(num): # gives our function a number parameter
    print(num + 100)

add_one_hundred(300) # This function needs an argument, I gave it 300, so
```

```
this will print out, 400

def add(x,y):
    print (x+y)

add(7,8) # Adds the 2 numbers together as it is taking 7 and x and 8 as y

def multiply(x,y):
    return x * y # Does not PRINT, this is calling back to 7*8

multiply(7,8)
print(multiply(7,8))

def  square_root(x):
    print(x ** .5)

square_root(64)

def nl():
    print ("\n") # prints a new line

nl()
```

Notes from Course

Function Definition:
A function is defined using the `def` keyword, followed by the function name, parentheses, and a colon. The function may also have parameters (optional) and a return statement (optional) to send back a result. Here's an example of a simple function definition:

```
def greet():
    print("Hello, World!")
```

Function Call:
To execute a function, you need to call it by its name, followed by parentheses. Here's an example of calling the `greet()` function:

```
greet()
```

Function Parameters:
Functions can accept parameters, which are variables that hold values passed to the function when it is called. Parameters allow you to customize the behavior of a function based on the values you provide. Here's an example of a function with parameters:

```
def greet(name):
    print("Hello, " + name + "!")

greet("Alice")
```

In the example above, the `greet()` function accepts a parameter named `name`. When the function is called with an argument, such as "Alice", the value is assigned to the `name` parameter within the function body.

Return Statement:
Functions can also return values using the `return` statement. The returned value can be assigned to a variable or used directly in expressions. Here's an example:

```
def add_numbers(a, b):
    return a + b


result = add_numbers(3, 4)
print(result)  # Output: 7
```

In this example, the `add_numbers()` function takes two parameters (`a` and `b`) and returns their sum. The returned value is then assigned to the `result` variable and printed.

Functions provide a way to encapsulate reusable code and improve the structure of your programs. They can take inputs, perform computations, and produce outputs, allowing you to modularize your code and make it more efficient and maintainable.

Boolean Expressions and Relational Operators

In Python, boolean expressions are expressions that evaluate to either `True` or `False`. They are typically used in conditional statements and logical operations to make decisions based on the truth or falsity of certain conditions. Relational operators are used to compare values and create boolean expressions.

```
# Boolean Expressions
print ("This section is about Boolean Expressions and Relational
Operators:")
print ("################################")
# This will mean basically true or false

bool1 = True
bool2 = 3*3 == 9 # this will mean this IS equal to this
bool3 = False
bool4 = 3*3 != 9 # this will mean this IS NOT equal to this
```

```
print (bool1,bool2,bool3,bool4)
print (type(bool1)) # This will return the class

bool5 = "True" # Putting it in quotes turn it into a string
print (type(bool5))

nl()
# Relational and Boolean Operators
greater_than = 7 > 5
less_than = 5 < 6
greater_then_equal_to = 7 >= 7
less_than_equal_to = 7 <= 7

test_and = (7 > 5) and (5 < 7) # Both statements are true, which makes this boolean, true
test_and2 = ( 7 > 5) and (5 < 7) # This is flase, because if one statement is false, it makes it false
test_or = (7 > 5) or (5 < 7) # This is true
test_or2 = (7 > 5) or (5 > 7) # True as only ONE of them needs to be true
```

NOT	True?
not False	True
not True	False

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	True?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

!=	True?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	True?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

Course Notes

Relational Operators:
Python provides several relational operators to compare values. Here are the commonly used relational operators:

Equality (`==`): Checks if two values are equal.
Inequality (`!=`): Checks if two values are not equal.
Greater than (`>`): Checks if the left value is greater than the right value.
Less than (`<`): Checks if the left value is less than the right value.
Greater than or equal to (`>=`): Checks if the left value is greater than or equal to the right value.
Less than or equal to (`<=`): Checks if the left value is less than or equal to the right value.

Boolean Expressions:
Boolean expressions are formed by combining relational expressions using logical operators. The logical operators in Python are:

Logical AND (`and`): Returns `True` if both operands are `True`.
Logical OR (`or`): Returns `True` if at least one operand is `True`.
Logical NOT (`not`): Negates the value of the operand.

Examples:

```
x = 5
y = 10


# Relational operators
print(x == y)   # Output: False
print(x < y)    # Output: True


# Boolean expressions
print(x < y and y > 0)    # Output: True
print(x < y or y < 0)     # Output: True
print(not (x == y))       # Output: True
```

In the example above, we have two variables `x` and `y`. We use the relational operators (`==` and `<`) to compare their values and create boolean expressions. The logical operators (`and`, `or`, and `not`) are then used to combine the relational expressions and evaluate the overall truth value.

Boolean expressions and relational operators are fundamental in controlling the flow of your program by making decisions based on conditions. They are extensively used in if statements, while loops, and other control structures to determine the execution path of your code.

Conditional Statements

In Python, conditional statements are used to perform different actions based on certain conditions. They allow you to control the flow of your program by executing specific blocks of code when certain conditions are met.

```
# Conditional Statements if/else - if/then/else
print ("This section is about conditional statements:")
print ("###########################################")

def drink(money):
    if money >= 2:
        return "You now have a drink!"
    else:
        return "You don't have a drink"
print(drink(3))
print(drink(1))

nl()
def alcohol(age,money):
    if (age >=21) and (money >=5):
        return "Getting turnt!"
    elif (age >= 21) and (money < 5):
        return "Go get mo money, mo money = mo problems tho"
    elif (age < 21) and (money >= 5):
        return "Nice try kid!"
    else:
        return "You're too young and too broke"
print (alcohol(21,5))
print (alcohol(21,4))
print (alcohol(20,5))
print (alcohol(20,4))
```

Course notes

if Statement:
The `if` statement is the most basic conditional statement. It executes a block of code if a given condition is true. Here's the general syntax:

```
if condition:
    # code to be executed if the condition is true
```

Example:

```
x = 5
if x > 0:
    print("x is positive")
```

In this example, the code inside the `if` block (`print("x is positive")`) will be executed if the condition `x > 0` is true.

if-else Statement:
The `if-else` statement allows you to specify two different blocks of code—one to be executed if the condition is true and another to be executed if the condition is false. Here's the syntax:

```
if condition:
    # code to be executed if the condition is true
else:
    # code to be executed if the condition is false
```

Example:

```
x = 5
if x > 0:
    print("x is positive")
else:
    print("x is not positive")
```

In this example, if `x` is greater than 0, the first block (`print("x is positive")`) will be executed. Otherwise, the second block (`print("x is not positive")`) will be executed.

if-elif-else Statement:
The if-elif-else statement allows you to check multiple conditions and execute different blocks of code based on those conditions. It provides more than two options for branching. Here's the syntax:

```
if condition1:
    # code to be executed if condition1 is true
elif condition2:
    # code to be executed if condition1 is false and condition2 is true
else:
    # code to be executed if all conditions are false
```

Example:

```
x = 5
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

In this example, if `x` is greater than 0, the first block will be executed. If it is less than 0, the second block will be executed. Otherwise, if none of the conditions are true, the third block will be executed.

Conditional statements allow you to make decisions and control the execution flow of your program based on certain conditions. They are essential for implementing logic and branching in your code.

Lists

In Python, a list is a versatile and mutable data structure that can hold a collection of items. It allows you to store multiple values of different data types in a single variable.
Lists are data structures, changable(mutable), able to be re-ordered, A group of elements. Everything in a list is an "item".

```
#Lists - Have brackets [ ]
print ("This section is about lists:")
print ("############################")
movies = ["Shaun of the Dead", "Hackers", "Ghost In The Shell", "Final Fantasy Advent Children"]

print (movies[1]) # Prints the second item in the list
print (movies[0]) # Prints the first item in the list
print (movies[1:3]) # Prints the first index number given until the last number given, but not include the last number
print (movies[1:]) # Prints from the first index number to the end
print (movies[:1]) # Prints everything from the end UP TO the first item
print (movies[-1]) # Prints the last item in the list
print (len(movies)) # Counts items in the list
movies.append("Hot Fuzz") # Appends an item to the end of the list
print (movies)
movies.insert(2, "Wall-E") # Inserts an item into the list at the given number
print (movies)
movies.pop() # Removes last item in the list
print (movies)
movies.pop(0) # Removes first item in the list
print (movies)

other_movies = ["Legally Blonde", "Mean girls"]
all_movies = movies + other_movies
print (all_movies)

grades = [["Th4ntis", 90], ["Lyra", 92], ["Space Cadet", 55]]
lyra_grade = grades[0][1] # Pulls the second item
print (lyra_grade)
grades[0][1] = 93
print (grades)
```

Course notes

List Creation:
To create a list, you enclose comma-separated values within square brackets `[ ]`. Here's an example:

```
fruits = ["apple", "banana", "orange"]
```

List Access:
You can access individual elements in a list using indexing. Indexing starts from 0 for the first element and goes up to the length of the list minus one. Here are some examples:

```
print(fruits[0])    # Output: "apple"
print(fruits[2])    # Output: "orange"
```

List Modification:
Lists are mutable, which means you can modify their elements. You can assign new values to specific positions in the list or use methods to modify the list itself. Here are some examples:

```
fruits[1] = "grape"     # Modifying an element
fruits.append("kiwi")   # Adding an element to the end
fruits.remove("apple")  # Removing an element
```

List Operations:
Python provides various operations that can be performed on lists. Some common operations include:

- Concatenation: You can use the `+` operator to concatenate two or more lists.
- Length: The `len()` function returns the number of elements in a list.
- Slicing: You can extract a sublist from a list using slicing.
- Iteration: You can use a loop to iterate over the elements of a list.
  Here are some examples:

```
fruits = ["apple", "banana", "orange"]
fruits2 = ["grape", "kiwi"]


combined = fruits + fruits2
print(combined)         # Output: ["apple", "banana", "orange", "grape",
"kiwi"]


print(len(fruits))      # Output: 3


sublist = fruits[1:3]
print(sublist)          # Output: ["banana", "orange"]
```

```
for fruit in fruits:
    print(fruit)        # Output: "apple", "banana", "orange"
```

Lists are powerful data structures in Python that allow you to store and manipulate collections of items. They are widely used for managing and processing data in various applications.

Tuples

In Python, a tuple is an ordered collection of elements, similar to a list. However, unlike lists, tuples are immutable, meaning their elements cannot be modified once they are created.

```
# TUPLES - Do not change ()
print ("This section is about tuples:")
print ("##############################")
grades = ("a", "b", "c", "d", "f")
print (grades[1])
```

Course Notes

Tuple Creation:
To create a tuple, you enclose comma-separated values within parentheses `( )`. Here's an example:

```
fruits = ("apple", "banana", "orange")
```

Tuple Access:
You can access individual elements in a tuple using indexing, similar to lists. Indexing starts from 0 for the first element. Here are some examples:

```
print(fruits[0])    # Output: "apple"
print(fruits[2])    # Output: "orange"
```

Tuple Immutability:
Tuples are immutable, meaning you cannot modify their elements. Once a tuple is created, its values cannot be changed. For example, attempting to assign a new value to an element will result in an error. Here's an example:

```
fruits[1] = "grape"    # This will raise an error
```

Tuple Operations:
Although tuples are immutable, you can perform certain operations on them:

- Concatenation: You can use the `+` operator to concatenate two or more tuples.
- Length: The `len()` function returns the number of elements in a tuple.
- Slicing: You can extract a subtuple from a tuple using slicing.

Here are some examples:

```
fruits = ("apple", "banana", "orange")
fruits2 = ("grape", "kiwi")


combined = fruits + fruits2
print(combined)         # Output: ("apple", "banana", "orange", "grape",
"kiwi")


print(len(fruits))      # Output: 3


subtuple = fruits[1:3]
print(subtuple)         # Output: ("banana", "orange")
```

Tuples are useful in situations where you want to store a collection
of values that should not be changed. They can be used to group related data elements and can also be used as keys in dictionaries. While tuples are immutable, they offer advantages such as faster performance and protection against unintentional modification.

Looping

In Python, looping allows you to repeat a block of code multiple times. It is a fundamental concept used for iterating over data structures, performing repetitive tasks, and controlling the flow of your program. There are two main types of loops in Python: `for` loops and `while` loops.

```
# LOOPING
print ("This section is about loops/looping:")
print ("####################################")
    # FOR Loops - Start to finish of an iterate
vegetables = ["cucumber", "tomato", "peppers"]
for x in vegetables:
    print (x) # Loops through the iterate

    # While Loops - Execute as long as TRUE
i = 1
while i < 10:
    print (i)
    i += 1 # Increases the value until i is 10
```

Course Notes

for Loop:
The `for` loop is used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object. It executes a block of code a fixed number of times, based on the elements or items in the sequence. Here's the general syntax:

```
for item in sequence:
    # code to be executed for each item in the sequence
```

Example:

```
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)
```

In this example, the `for` loop iterates over each item in the `fruits` list, and the block of code inside the loop (`print(fruit)`) is executed for each item. It will output:

```
apple
banana
orange
```

while Loop:
The `while` loop is used to repeatedly execute a block of code as long as a given condition is true. It continues looping until the condition becomes false. Here's the general syntax:

```
while condition:
    # code to be executed while the condition is true
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

In this example, the `while` loop will continue executing the code inside the loop (`print(count)`) as long as the condition `count < 5 is` true. It will output:

```
0
1
2
3
4
```

The `break` and `continue` Statements:
Within loops, you can use the `break` statement to exit the loop prematurely and the `continue`

statement to skip the current iteration and move to the next one.

Looping provides a powerful mechanism for iterating over data, performing calculations, and controlling program flow. By using loops effectively, you can automate repetitive tasks and process large amounts of data efficiently.

Advances Strings

In Python, the str (string) data type represents a sequence of characters enclosed within single quotes (`' '`) or double quotes (`" "`). Strings in Python are immutable, which means they cannot be changed after they are created.

```
# ADVANCED STRINGS
print ("This section is about Advanced Strings:")
print ("########################################")
my_name = "Th4ntis"
print (my_name[0]) # grabs first letter
print (my_name[-1]) # Grabs last letter

sentence = "This is a sentence"
print (sentence[:4])
print (sentence.split()) # delimeter - takes something and split based on
that - default is a space

sentence_split = sentence.split()
sentence_join = "-".join(sentence_split) # Adds a - between the words we are
pulling out
print (sentence_join)

quote = "He screamed 'Hack the planet!'" # How to use quotes, can be as is
or swap the single/double quote
print (quote)
quote = "He screamed \"Hack the planet!\"" # This is using a character
escape
print (quote)

too_much_space = "                        hello                        "
print (too_much_space)
print (too_much_space.strip()) # uses space as a delimeter and removes it

print ("A" in "Apple") # Returns True
print ("a" in "Apple") # Returns false due to lowercase A
```

```
letter = "A"
word = "Apple"
print (letter.lower() in word.lower()) # Improved way

movie = "Hackers"
print ("My favorite movie is " + movie) # Old way
print ("My favorite movie is {}.".format(movie)) # Using string format
method
print ("My favorite movie is %s" % movie)
print (f"My favorite movie is  {movie}.")
```

Course notes

Creation:

Example: `my_string = 'Hello, World!'` or `my_string = "Hello, World!"`

Accessing Characters:
You can access individual characters within a string using indexing, starting from 0.
Example: `print(my_string[0])` would output 'H'.

String Concatenation:
You can concatenate (join) two or more strings using the + operator.
Example: `greeting = 'Hello' + ' ' + 'World!'` would result in 'Hello World!'.

String Length:
The `len()` function can be used to determine the length (number of characters) of a string.
Example: `print(len(my_string))` would output the length of the string.
String Slicing:

You can extract a substring from a string using slicing, specifying the start and end indices.
Example: `substring = my_string[7:12]` would extract the substring 'World'.

String Methods:
Python provides various built-in methods to manipulate and transform strings. Examples include `upper(), lower(), strip(), split(), replace()`, and more.
Example: `print(my_string.upper())` would output 'HELLO, WORLD!'.

String Formatting:
String formatting allows you to embed values within a string. This can be done using the `%` operator or the `format()` method.
Example:

```
name = 'Alice'
age = 30
```

```
print("My name is %s and I'm %d years old." % (name, age))
# Output: My name is Alice and I'm 30 years old.
```

These are just a few key concepts related to strings in Python. Strings in Python are versatile and support a wide range of operations and manipulations.

Dictionaries

In Python, a dictionary is an unordered collection of key-value pairs. It is a versatile and powerful data structure that allows you to store, retrieve, and manipulate data based on unique keys.

```
# DICTIONARIES - Key/value pairs {}
print ("This section is about Dictionaries:")
print ("###################################")
drinks = {"Water": 2, "Orange Juice": 3, "Chocolate Milk": 4} # the drink is the key, the price is the value
print (drinks)

employees = {"Finance": ["Bob", "Linda", "Tina"], "IT": ["Gene", "Louise", "Teddy"], "HR": ["Mort"]}
print (employees)
employees ["Legal"] = ["Frond"] # adds new key:value pair
print (employees)

employees.update({"Sales": ["Andie", "Ollie"]}) # adds new key:value pair
print (employees)

drinks["Water"] = 4 # updates value on item in dictionary
print (drinks.get("Water")) # Pulls the value of the key in the dictionary
```

Course notes

Dictionary Creation:

To create a dictionary, you enclose key-value pairs within curly braces `{ }`, separating each pair with a colon `:`. Here's an example:

```
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science"
}
```

Dictionary Access:
You can access the values in a dictionary by referring to their corresponding keys. Keys provide a way

to uniquely identify and retrieve values. Here's an example:

```
print(student["name"])    # Output: "Alice"
print(student["age"])     # Output: 20
```

Dictionary Modification:
Dictionaries are mutable, which means you can modify their values by assigning new values to specific keys. Here's an example:

```
student["age"] = 21       # Modifying a value
student["city"] = "London"    # Adding a new key-value pair
```

Dictionary Operations:
Python provides various operations that can be performed on dictionaries. Some common operations include:

- Length: The len() function returns the number of key-value pairs in a dictionary.
- Iteration: You can iterate over the keys, values, or key-value pairs of a dictionary using loops.
- Deletion: You can remove a key-value pair from a dictionary using the del keyword.

Here are some examples:

```
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science"
}


print(len(student))          # Output: 3


for key in student:
    print(key, student[key])  # Output: "name Alice", "age 20", "major
Computer Science"


del student["age"]           # Deleting a key-value pair
```

Dictionaries are powerful data structures that provide a flexible way to store and retrieve data based on keys. They are commonly used for organizing and manipulating data that requires quick and efficient access.

Importing Modules

In Python, importing modules allows you to access and use code that resides in external Python files or libraries. Modules are a way to organize and reuse code, making it easier to manage and maintain large projects.

```
# IMPORTING MODULES - Built in but not available unless we import them
import sys # imports system functions and parameters
from datetime import datetime as dt # gives datetime an alias


print (sys.version) # Prints python version
# print (datetime.now()) # Prints time
print (dt.now())
```

Course notes

Importing Entire Modules:
To import an entire module, you use the `import` keyword followed by the module name. Here's an example:

```
import math

result = math.sqrt(25)
print(result)   # Output: 5.0
```

In this example, the `math` module is imported, and the `sqrt()` function from the module is used to calculate the square root of 25.

Importing Specific Functions or Variables:
If you only need to use specific functions or variables from a module, you can import them directly. Here's an example:

```
from math import sqrt

result = sqrt(25)
print(result)   # Output: 5.0
```

In this case, only the `sqrt()` function is imported from the `math` module, so you can use it directly without prefixing it with the module name.

Importing Modules with an Alias:
You can also import a module and give it an alias using the `as` keyword. This can be helpful when dealing with modules with long names or to avoid naming conflicts. Here's an example:

```
import math as m

```

```
result = m.sqrt(25)
print(result)   # Output: 5.0
```

In this example, the `math` module is imported and assigned the alias `m`, so you can use `m.sqrt()` instead of `math.sqrt()`.

Importing All Functions and Variables:
If you want to import all functions and variables from a module, you can use the `*` wildcard character. However, it is generally recommended to import only what you need to avoid namespace pollution. Here's an example:

```
from math import *

result = sqrt(25)
print(result)   # Output: 5.0
```

In this case, all functions and variables from the `math` module are imported directly, allowing you to use them without prefixing with the module name.

Importing modules enables you to access and utilize a wide range of functionality provided by the Python standard library or third-party libraries. It promotes code reusability, modularity, and maintainability in your Python programs.

Sockets

In Python, sockets are a fundamental networking concept used for communication between computers over a network. Sockets enable programs to establish connections, send data, and receive data over various network protocols, such as TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

```
#SOCKETS - Connects nodes together

import socket

HOST = "127.0.0.1"
PORT = 7777

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #AF_INET is IPv4,
sock_stream is a port

s.connect((HOST, PORT))
```

Course Notes

Socket Creation:
To use sockets in Python, you need to import the `socket` module. You can create a socket object using the `socket.socket()` function, which takes two parameters: the address family (e.g., `socket.AF_INET` for IPv4) and the socket type (e.g., `socket.SOCK_STREAM` for TCP or `socket.SOCK_DGRAM` for UDP). Here's an example:

```
import socket


# Create a TCP socket
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


# Create a UDP socket
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Socket Communication:
Once you have a socket object, you can use various methods to establish connections, send data, and receive data. Here are some commonly used methods:

- `socket.connect(address)`: Establishes a connection to a remote address.
- `socket.bind(address)`: Binds the socket to a specific address and port.
- `socket.listen(backlog)`: Listens for incoming connections on a TCP socket.
- `socket.accept()`: Accepts an incoming connection and returns a new socket object for communication.
- `socket.send(data)`: Sends data over the socket.
- `socket.recv(buffer_size)`: Receives data from the socket.

Here's an example of a basic TCP server that listens for incoming connections:

```
import socket


# Create a TCP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


# Bind the socket to a specific address and port
server_address = ('localhost', 1234)
server_socket.bind(server_address)


# Listen for incoming connections
```

```
server_socket.listen(5)


while True:
    # Accept a client connection
    client_socket, client_address = server_socket.accept()


    # Receive and send data
    data = client_socket.recv(1024)
    client_socket.send(b"Received: " + data)


    # Close the client socket
    client_socket.close()
```

Socket programming in Python allows you to create client-server applications, networked applications, and perform various networking tasks. It provides a powerful and flexible way to communicate over networks using different protocols. The `socket` module in Python provides a wide range of functions and methods to handle network communication efficiently.

Building a Port Scanner

```
#!/bin/python3

# Usage: python3 scanner.py <IP>

import sys
import socket
from datetime import datetime

# Define our target

if len(sys.argv) == 2:
        target = socket.gethostbyname(sys.argv[1]) # Translates hostname to
IPv4, argv is the number of arguments we are going to be giving
else:
        print ("Invalid amount of argument.")
        print ("Syntax: python3 scanner.py <ip>")

# Adds a pretty banner
print ("-" * 40)
print ("Scanning target: " + target)
```

```
print ("Time started: " + str(datetime.now()))
print ("-" * 40)

try:
        for port in range(50,85): # This is not threaded and can take a long time, so we will scan port 50-85
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Sets variable for the IP address and port we will be trying to connect to
                socket.setdefaulttimeout (1) # Sets a default time out of 1, if it doesn't respond after 1 second, it moves on
                result = s.connect_ex((target,port)) # Saying I want to connect to target on port. The _ex is an error indicator, if a port is open, the error result returns 0, if a port is closed, it return a 1.
                if result == 0:
                        print (f"Port {port} is open")
                s.close() # if the port is closed, it will close the connection and move on

except KeyboardInterrupt:
        print ("\nExiting program...")
        sys.exit()

except socket.gaierror:
        print ("Hostname could not be resolved.")
        sys.exit()

except socket.error:
        print ("Could not connect to host/IP")
        sys.exit()
```

User Input

In Python, you can interact with the user and receive input using the `input()` function. The `input()` function allows you to prompt the user for input and receive the input as a string.

```
# USER INPUT

# Uncomment this section to see basic input
# name = input ("What is your name?: ")
# drink = input ("What do you prefer to drink?: ")
# print ("Hello " + name + "! Have some "+ drink + ".")

# Uncomment this section to see Calculator input
```

```
o = input("Give me an operation(+, -, *, /, **): ")
x = float(input ("Give me a number: "))
y = float(input ("Give me another number: "))

if o == "+":
        print (x + y)
elif o == "-":
        print (x - y)
elif o == "/":
        print (x / y)
elif o == "*":
        print (x * y)
elif o == "**" or o == "^":
        print (x ** y)
else:
        print ("Unknown operation.")
```

Course Notes

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

In this example, the `input()` function is used to prompt the user to enter their name. The message "Enter your name: " is displayed to the user as a prompt. The user can then type their name and press Enter. The input provided by the user is stored in the variable `name`, and the program prints a greeting using the entered name.

The `input()` function always returns the user's input as a string. If you need to convert the input to a different data type, such as an integer or float, you can use appropriate conversion functions like `int()` or `float()`.

```
age = input("Enter your age: ")
age = int(age)  # Convert the input to an integer

print("You will be " + str(age + 1) + " next year.")
```

In this example, the user is asked to enter their age. The input is stored as a string in the variable `age`. To perform arithmetic calculations, the input is converted to an integer using the int() function. The program then adds 1 to the age and prints the result.

When using user input, keep in mind that it is a string by default. Ensure proper validation and error handling if you expect specific data types or want to handle invalid input.

User input allows you to make your programs interactive and dynamic by accepting input from users during runtime. It provides a way to customize program behavior based on user responses.

Reading and Writing Files

In Python, you can read from and write to files using file objects and various methods provided by the built-in `open()` function.

```
# READING FILES

# months = open("months.txt")
# print (months.read()) # Reads the file and outputs it to the terminal
# print (months.readline()) # Reads the file line by line
# print (months.readline()) # Reads the file line by line
# print (months.readlines()) # Outputs all lines as an array
# print (months.mode) # Tells us which mode we are in, readable, writable,
etc
# print (months.readable()) # Tells us TRUE or FALSE if the file is readable


# for month in months: # For loop to read each line
#       print(month.strip()) # .strip() removes each new empty line

# months.close() # Gracefully closes the file

# WRITING FILES
# days = open("days.txt", "w") # Writes to a file, overwriting content
days = open("days.txt", "a") # Writes to a file, appending content

days.write("Monday") # Adds the string to the file
days.write("\nTuesday")

days.close() # Gracefully closes file
```

Course Notes

Reading Files:
To read from a file, you need to open it in read mode using the open() function. Once the file is open, you can use methods like read(), readline(), or readlines() to retrieve the contents of the file.

- `read()`: Reads the entire content of the file as a string.
- `readline()`: Reads a single line from the file.
- `readlines()`: Reads all lines from the file and returns them as a list.

Here's an example of reading from a file:

```
# Open the file in read mode
file = open("example.txt", "r")

# Read the entire content
content = file.read()
print(content)

# Read a single line
line = file.readline()
print(line)

# Read all lines
lines = file.readlines()
print(lines)

# Close the file
file.close()
```

Writing Files:
To write to a file, you need to open it in write mode using the `open()` function. Once the file is open, you can use the `write()` method to write content to the file.

- `write(content)`: Writes the specified content to the file.

Here's an example of writing to a file:

```
# Open the file in write mode
file = open("example.txt", "w")

# Write content to the file
file.write("Hello, World!\n")
file.write("This is a new line.")

# Close the file
file.close()
```

Appending to Files:
To append content to an existing file without overwriting its existing contents, you can open the file in append mode (`"a"`) using the `open()` function. Then, you can use the `write()` method to append content to the file.

```
# Open the file in append mode
file = open("example.txt", "a")
```

```
# Append content to the file
file.write("\nThis is appended content.")



# Close the file
file.close()
```

It is generally recommended to use the `with` statement when working with files. This ensures that the file is properly closed even if an exception occurs.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Reading and writing files in Python allows you to handle external data, process large amounts of information, and store program outputs for later use. It is important to properly manage file resources and close them after use to avoid memory leaks and ensure data integrity.

Classes and Objects

In Python, classes and objects are key components of object-oriented programming (OOP). They provide a way to structure code and define custom data types.

```
class Employees:

    def __init__(self, name, department, role, salary, years_employed):
        self.name = name
        self.department = department
        self.role = role
        self.salary = salary
        self.years_employed = years_employed

    def eligible_for_retirement(self):
        if self.years_employed >= 20:
            return True
        else:
            return False
```

```
#!/bin/python3

from Employees import Employees
e1 = Employees("Bob", "Sales", "Directory of Sales", 100000, 20)
e2 = Employees("Linda", "Executive", "CIO", 150000, 10)
```

```
print (e1.name)
print (e2.role)
print (e1.eligible_for_retirement())
```

Course Notes

Classes:
A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will possess. You can think of a class as a blueprint for creating instances of objects with similar characteristics and functionalities. Here's an example of a simple class definition:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")

    def display_info(self):
        print("Name:", self.name)
        print("Age:", self.age)
```

In this example, the `Dog` class has attributes `name` and `age`, and methods `bark()` and `display_info()`. The `__init__()` method is a special method known as the constructor, which is called when an object of the class is created.

Objects:
An object is an instance of a class. It is created based on the blueprint provided by the class. Each object has its own set of attributes and can invoke the methods defined in the class. You create objects by calling the class as if it were a function. Here's an example:

```
# Create objects of the Dog class
dog1 = Dog("Buddy", 5)
dog2 = Dog("Max", 3)

# Call methods on the objects
dog1.bark()               # Output: "Woof!"
dog1.display_info()       # Output: "Name: Buddy", "Age: 5"

dog2.bark()               # Output: "Woof!"
dog2.display_info()       # Output: "Name: Max", "Age: 3"
```

In this example, `dog1` and `dog2` are objects created from the `Dog` class. Each object has its own set of attributes (`name` and `age`) and can invoke the methods (`bark()` and `display_info()`) defined in the class.

Classes and objects are essential in object-oriented programming as they provide a way to organize code, encapsulate data, and define reusable entities. They enable you to model real-world entities, create custom data types, and build complex systems by leveraging the principles of inheritance, polymorphism, and encapsulation.

Building a Shoe Budget Tool

```
class drinks:
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)

        def budget_check(self, budget):
                if not isinstance(budget, (int, float)):
                        print('Invalid entry. Please enter a number.')
                        exit()

        def change(self, budget):
                return (budget - self.price)

        def buy(self, budget):
                self.budget_check(budget)

                if budget >= self.price:
                        print(f'You can buy the {self.name}')

                        if budget == self.price:
                                print('You have exactly enough money for
this drink.')
                        else:
                                print(f'You can buy the drink and have
${self.change(budget)} left over')

                        exit('Thanks for using our budget app!')
```

```
#!/bin/python3

from drinks import drinks
```

```
normal = ("Water", 2)
caffeine = ("Soda", 4)
energy = ("Energy Drink", 6)

try:
    drinks_budget = float(input("What is your budget?: "))
except ValueError:
    exit("Please enter a number.")

for drinks in [energy, caffeine, normal]:
    drinks.buy(drinks_budget)
```

Course Notes

```
class Shoes:
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)

        def budget_check(self, budget):
                if not isinstance(budget, (int, float)):
                        print('Invalid entry. Please enter a number.')
                        exit()

        def change(self, budget):
                return (budget - self.price)

        def buy(self, budget):
                self.budget_check(budget)

                if budget >= self.price:
                        print(f'You can cop some {self.name}')

                        if budget == self.price:
                                print('You have exactly enough money for
these shoes.')
                        else:
                                print(f'You can buy these shoes and have
${self.change(budget)} left over')

                        exit('Thanks for using our shoe budget app!')
```

```
#!/bin/python3
```

```
from Shoes import Shoes

low = Shoes('And 1s', 30)
medium = Shoes('Air Force 1s', 120)
high = Shoes('Off Whites', 400)

try:
   shoe_budget = float(input('What is your shoe budget? '))
except ValueError:
   exit('Please enter a number')

for shoes in [high, medium, low]:
   shoes.buy(shoe_budget)
```

```
months = open('months.txt')

print(months)
print(months.mode)
print(months.readable())

months.close()



print(months.read())
print(months.readline()) #reads one line
print(months.readline()) #reads next line
print(months.readlines()) #prints an array
print(months.readlines()) #prints an empty array because we already read it
months.seek(0)
print(months.readlines()) - prints an array again

months.seek(0)
for month in months:
    print(month)

months.seek(0)
for month in months:
    print(month.strip())

days = open("days.txt", "w")
days.write("Monday")
days.close()
```

```
days = open("days.txt", "w")
days.write("\nTuesday") - overwrites
days.close()

days = open("days.txt", "a")
days.write("\nWednesday") - appends
days.close()
```