ive basic syntax in python in tables one line

ChatGPT

Here's a table with basic Python syntax in a concise one-liner format:

| **Concept** | **Syntax** |
| --- | --- |
| **Variable Assignment** | x = 10 |
| **Print Statement** | print("Hello, World!") |
| **If Statement** | if x > 5: print("x is greater than 5") |
| **For Loop** | for i in range(5): print(i) |
| **While Loop** | while x > 0: x -= 1 |
| **Function Definition** | def func\_name(): pass |
| **List Definition** | my\_list = [1, 2, 3] |
| **Dictionary Definition** | my\_dict = {"key": "value"} |
| **Class Definition** | class MyClass: pass |
| **Importing Modules** | import math |
| **Comment** | # This is a comment |
| **String Concatenation** | greeting = "Hello, " + "World!" |
| **List Comprehension** | [x\*\*2 for x in range(10)] |
| **Lambda Function** | square = lambda x: x\*\*2 |
| **Try-Except Block** | try: pass except: pass |

4o

next similarly give about variables and datatypes

ChatGPT

Here's a table summarizing Python variables and data types in a concise one-liner format:

| **Concept** | **Syntax** |
| --- | --- |
| **Integer Variable** | x = 10 |
| **Float Variable** | y = 3.14 |
| **String Variable** | name = "John" |
| **Boolean Variable** | is\_active = True |
| **List Variable** | fruits = ["apple", "banana"] |
| **Tuple Variable** | coordinates = (10, 20) |
| **Dictionary Variable** | person = {"name": "Alice", "age": 30} |
| **Set Variable** | unique\_nums = {1, 2, 3} |
| **NoneType Variable** | nothing = None |
| **Type Checking** | type(x) |
| **Multiple Assignment** | a, b, c = 1, 2, 3 |
| **String Formatting** | greeting = f"Hello, {name}" |
| **Convert to String** | str(100) |
| **Convert to Integer** | int("42") |
| **Convert to Float** | float("3.14") |

4o

explain this pgm: >>> def fib(n): >>> a, b = 0, 1 >>> while a < n: >>> print(a, end=' ') >>> a, b = b, a+b >>> print() >>> fib(1000)

ChatGPT

This program defines and executes a function to generate and print the Fibonacci sequence up to a specified number. Let's break it down step by step:

**Code:**

python

Copy code

def fib(n):

a, b = 0, 1

while a < n:

print(a, end=' ')

a, b = b, a + b

print()

fib(1000)

**Explanation:**

1. **Function Definition**:
   * def fib(n): defines a function named fib that takes a single parameter n.
2. **Variable Initialization**:
   * a, b = 0, 1: Two variables a and b are initialized. Here, a is set to 0, and b is set to 1. These will be the first two numbers in the Fibonacci sequence.
3. **While Loop**:
   * while a < n:: This loop runs as long as the value of a is less than n. It ensures that the sequence continues until a reaches or exceeds n.
   * Inside the loop:
     + print(a, end=' '): This prints the current value of a followed by a space, without moving to a new line (due to end=' ').
     + a, b = b, a + b: This is a simultaneous assignment. The value of b is assigned to a, and the sum of the previous a and b is assigned to b. This effectively shifts the Fibonacci sequence forward.
4. **Final Print**:
   * print(): After the loop completes (when a is no longer less than n), this prints a newline to ensure that the output ends neatly.
5. **Function Call**:
   * fib(1000): This calls the fib function with 1000 as the argument, meaning the function will print all Fibonacci numbers less than 1000.

**Output:**

When you run this program, it prints the Fibonacci sequence up to 1000:

Copy code

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

Each number in the sequence is the sum of the two preceding ones, starting from 0 and 1. The sequence continues until the next number would be 1597, which exceeds 1000, so the loop stops.

4o

how to understand this a, b = b, a + b: This is a simultaneous assignment.

ChatGPT

The line a, b = b, a + b in Python is an example of a **simultaneous assignment** or **tuple unpacking**. Here's how it works:

**Understanding Simultaneous Assignment:**

* **Simultaneous Assignment** means that the right-hand side expressions are evaluated first, and then their values are assigned to the left-hand side variables all at once. This allows you to swap or update variables in a single line without needing a temporary variable.

**Step-by-Step Breakdown:**

Let’s say a = 2 and b = 3 before the assignment:

1. **Right-Hand Side Evaluation**:
   * b, a + b: The right-hand side is a tuple (b, a + b).
   * Here, b evaluates to 3, and a + b evaluates to 2 + 3 = 5.
2. **Tuple Creation**:
   * A tuple is created with the values (3, 5).
3. **Assignment**:
   * The first value of the tuple, 3, is assigned to a.
   * The second value of the tuple, 5, is assigned to b.

**Visual Example:**

Before the assignment:

* a = 2
* b = 3

After a, b = b, a + b:

* a is now 3
* b is now 5

So, the assignment effectively moves the values forward in the Fibonacci sequence:

* a takes on the old value of b.
* b becomes the sum of the old values of a and b.

**Why is This Useful?**

In the context of the Fibonacci sequence:

* You need to shift the current numbers a and b to the next numbers in the sequence.
* a should take the current value of b (the next number in the sequence).
* b should become the sum of a and b (which represents the next Fibonacci number).

This simultaneous assignment allows for this shift in a single, concise line of code, making it both efficient and readable.

4o

what all are these print options available in python

ChatGPT

In Python, the print() function has several options and parameters to customize how output is displayed. Here’s a summary of the available options:

**Basic Usage**

python

Copy code

print(object, ...)

* **object**: The value(s) to be printed. You can pass multiple objects separated by commas.

**Parameters**

1. **sep**: Specifies the separator between objects. Default is a single space.

python

Copy code

print("apple", "banana", "cherry", sep=", ")

# Output: apple, banana, cherry

1. **end**: Specifies what to print at the end of the line. Default is a newline (\n).

python

Copy code

print("Hello", end=" ")

print("World")

# Output: Hello World

1. **file**: Specifies a file-like object where the output should be directed. Default is sys.stdout (the console).

python

Copy code

with open('output.txt', 'w') as f:

print("Hello, file!", file=f)

# Writes "Hello, file!" to output.txt

1. **flush**: A boolean that specifies whether to forcibly flush the stream. Default is False.

python

Copy code

import time

print("Processing...", end="", flush=True)

time.sleep(2)

print(" Done!")

# Output: Processing... (with "Done!" appearing after a 2-second delay)

**Examples**

1. **Custom Separator**:

python

Copy code

print("one", "two", "three", sep=" - ")

# Output: one - two - three

1. **Custom End Character**:

python

Copy code

print("Hello", end="!")

print("World")

# Output: Hello!World

1. **Print to a File**:

python

Copy code

with open('log.txt', 'w') as file:

print("Log entry", file=file)

# Writes "Log entry" to log.txt

1. **Flush the Output**:

python

Copy code

print("Starting process...", end="", flush=True)

# Immediate output to the console without buffering

These options give you control over formatting the output, redirecting it to different destinations, and managing buffering behavior.

4o mini

give tabular about operators in pthon

ChatGPT

Here's a table summarizing the various operators available in Python:

| **Operator Type** | **Operator** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Arithmetic Operators** | + | Addition | 3 + 2 yields 5 |
|  | - | Subtraction | 5 - 3 yields 2 |
|  | \* | Multiplication | 4 \* 2 yields 8 |
|  | / | Division | 8 / 2 yields 4.0 |
|  | // | Floor Division (returns quotient without remainder) | 8 // 3 yields 2 |
|  | % | Modulus (remainder of division) | 8 % 3 yields 2 |
|  | \*\* | Exponentiation (power) | 2 \*\* 3 yields 8 |
| **Comparison Operators** | == | Equality | 5 == 5 yields True |
|  | != | Inequality | 5 != 3 yields True |
|  | > | Greater than | 7 > 5 yields True |
|  | < | Less than | 3 < 5 yields True |
|  | >= | Greater than or equal to | 5 >= 5 yields True |
|  | <= | Less than or equal to | 3 <= 5 yields True |
| **Logical Operators** | and | Logical AND | (5 > 3) and (4 > 2) yields True |
|  | or | Logical OR | (5 > 3) or (4 < 2) yields True |
|  | not | Logical NOT | not (5 > 3) yields False |
| **Assignment Operators** | = | Assignment | x = 10 sets x to 10 |
|  | += | Addition assignment | x += 5 (equivalent to x = x + 5) |
|  | -= | Subtraction assignment | x -= 5 (equivalent to x = x - 5) |
|  | \*= | Multiplication assignment | x \*= 5 (equivalent to x = x \* 5) |
|  | /= | Division assignment | x /= 5 (equivalent to x = x / 5) |
|  | //= | Floor Division assignment | x //= 5 (equivalent to x = x // 5) |
|  | %= | Modulus assignment | x %= 5 (equivalent to x = x % 5) |
|  | \*\*= | Exponentiation assignment | x \*\*= 2 (equivalent to x = x \*\* 2) |
| **Bitwise Operators** | & | Bitwise AND | 5 & 3 yields 1 |
|  | ` | ` | Bitwise OR |
|  | ^ | Bitwise XOR (exclusive OR) | 5 ^ 3 yields 6 |
|  | ~ | Bitwise NOT (one's complement) | ~5 yields -6 |
|  | << | Bitwise left shift | 5 << 1 yields 10 |
|  | >> | Bitwise right shift | 5 >> 1 yields 2 |
| **Identity Operators** | is | Identity (checks if two variables point to the same object) | x is y |
|  | is not | Identity negation | x is not y |
| **Membership Operators** | in | Membership test (checks if a value is in a sequence) | 5 in [1, 2, 3, 4, 5] yields True |
|  | not in | Membership test negation | 5 not in [1, 2, 3, 4] yields True |

This table covers the primary operators used in Python for various types of operations.

4o mini

similarly give about control flow

ChatGPT

Here's a table summarizing the control flow statements in Python:

| **Control Flow Statement** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **If Statement** | if condition: # code | Executes the code block if the condition is True. | if x > 0: print("Positive") |
| **If-Else Statement** | if condition: # code else: # code | Executes the first block if the condition is True, otherwise the second block. | if x > 0: print("Positive") else: print("Non-positive") |
| **If-Elif-Else Statement** | if condition: # code elif condition: # code else: # code | Evaluates multiple conditions in sequence, executing the corresponding block. | if x > 0: print("Positive") elif x < 0: print("Negative") else: print("Zero") |
| **While Loop** | while condition: # code | Repeats the code block as long as the condition is True. | while x < 5: x += 1 print(x) |
| **For Loop** | for variable in iterable: # code | Iterates over each item in the iterable (e.g., list, tuple). | for i in range(5): print(i) |
| **Break Statement** | break | Exits the nearest enclosing loop prematurely. | for i in range(10): if i == 5: break |
| **Continue Statement** | continue | Skips the rest of the code inside the nearest enclosing loop for the current iteration. | for i in range(10): if i % 2 == 0: continue print(i) |
| **Pass Statement** | pass | A null statement used as a placeholder; does nothing. | if x > 0: pass |
| **Try-Except Block** | try: # code except Exception as e: # code | Handles exceptions (errors) that occur in the try block. | try: print(1 / 0) except ZeroDivisionError as e: print("Cannot divide by zero") |
| **Try-Except-Finally Block** | try: # code except Exception as e: # code finally: # code | Executes finally code regardless of whether an exception was raised or not. | try: f = open("file.txt", "r") except IOError: print("File not found") finally: f.close() |
| **Raise Exception** | raise Exception("message") | Raises an exception manually. | raise ValueError("Invalid value") |

This table covers the primary control flow statements used in Python for managing program execution.

4o mini

similarly give about string handling

ChatGPT

Here's a table summarizing common string handling operations and methods in Python:

| **Operation/Method** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **String Concatenation** | str1 + str2 | Concatenates str1 and str2. | "Hello" + " World" yields "Hello World" |
| **String Repetition** | str \* n | Repeats str n times. | "Ha" \* 3 yields "HaHaHa" |
| **String Length** | len(str) | Returns the length of the string. | len("Hello") yields 5 |
| **Convert to Uppercase** | str.upper() | Converts all characters to uppercase. | "hello".upper() yields "HELLO" |
| **Convert to Lowercase** | str.lower() | Converts all characters to lowercase. | "HELLO".lower() yields "hello" |
| **Title Case** | str.title() | Converts the first character of each word to uppercase. | "hello world".title() yields "Hello World" |
| **Capitalize** | str.capitalize() | Converts the first character to uppercase and the rest to lowercase. | "hello world".capitalize() yields "Hello world" |
| **Strip Whitespace** | str.strip() | Removes leading and trailing whitespace. | " hello ".strip() yields "hello" |
| **Replace Substring** | str.replace(old, new) | Replaces occurrences of old with new. | "hello world".replace("world", "there") yields "hello there" |
| **Split String** | str.split(separator) | Splits the string into a list using the specified separator. | "a,b,c".split(",") yields ["a", "b", "c"] |
| **Join List** | separator.join(list) | Joins elements of a list into a string with separator. | ",".join(["a", "b", "c"]) yields "a,b,c" |
| **Find Substring** | str.find(sub) | Returns the lowest index where sub is found. Returns -1 if not found. | "hello".find("e") yields 1 |
| **Check if Substring Exists** | sub in str | Checks if sub is in str. | "hello" in "hello world" yields True |
| **Check Start of String** | str.startswith(prefix) | Checks if the string starts with prefix. | "hello".startswith("he") yields True |
| **Check End of String** | str.endswith(suffix) | Checks if the string ends with suffix. | "hello".endswith("lo") yields True |
| **String Formatting** | f"{} {}" or str.format() | Formats strings using placeholders. | f"Hello {name}" or "Hello {}".format(name) |
| **Escape Characters** | str.encode(encoding) | Encodes the string using the specified encoding. | "hello".encode("utf-8") yields b'hello' |
| **String Comparison** | str1 == str2 | Compares two strings for equality. | "abc" == "abc" yields True |

This table covers the primary string handling methods and operations available in Python for manipulating and working with strings.

4o mini

similarly give about functions

ChatGPT

Here's a table summarizing common aspects of functions in Python:

| **Aspect/Operation** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Function Definition** | def function\_name(parameters): # code | Defines a function with a name and parameters. | def greet(name): print("Hello, " + name) |
| **Function Call** | function\_name(arguments) | Calls a function with specified arguments. | greet("Alice") |
| **Default Arguments** | def function\_name(param1, param2=default): | Defines default values for parameters. | def greet(name, msg="Hello"): print(msg + ", " + name) |
| **Keyword Arguments** | function\_name(arg1=value1, arg2=value2) | Passes arguments by name, allowing for out-of-order arguments. | greet(name="Bob", msg="Hi") |
| **Arbitrary Arguments** | def function\_name(\*args): | Allows the function to accept any number of positional arguments. | def add(\*numbers): return sum(numbers) |
| **Arbitrary Keyword Arguments** | def function\_name(\*\*kwargs): | Allows the function to accept any number of keyword arguments. | def print\_info(\*\*info): print(info) |
| **Return Value** | return value | Returns a value from the function. | def square(x): return x \* x |
| **Lambda Function** | lambda parameters: expression | Defines a small anonymous function. | square = lambda x: x \* x |
| **Function Documentation** | """docstring""" | Provides documentation for the function. | def add(x, y): """Returns the sum of x and y.""" |
| **Nested Functions** | def outer\_function(): def inner\_function(): | Functions defined within other functions. | def outer(): def inner(): return "inner" return inner() |
| **Function Scope** | Variables defined inside a function are local to that function. | Scope of variables within functions. | def func(): x = 10 print(x) func() |
| **Function Arguments** | Functions can accept different types of arguments including positional, default, keyword, and arbitrary. |  |  |

This table provides a concise overview of key function-related features and operations in Python, covering definition, calling, and various argument handling techniques.

4o mini

similarly give about modules and packages

ChatGPT

Here's a table summarizing key concepts and operations related to modules and packages in Python:

| **Concept/Operation** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Import a Module** | import module\_name | Imports a module to use its functions and variables. | import math |
| **Import Specific Function** | from module\_name import function\_name | Imports a specific function from a module. | from math import sqrt |
| **Import All Functions** | from module\_name import \* | Imports all functions and variables from a module. | from math import \* |
| **Module Alias** | import module\_name as alias | Imports a module with an alias for convenience. | import numpy as np |
| **Create a Module** | # filename: module\_name.py def function\_name(): | Defines a module by creating a Python file with functions. | def greet(name): print("Hello, " + name) |
| **Using a Module** | module\_name.function\_name() | Calls a function from a module. | math.sqrt(16) |
| **Check if Module is Main** | if \_\_name\_\_ == "\_\_main\_\_": # code | Checks if the module is being run directly. | if \_\_name\_\_ == "\_\_main\_\_": main() |
| **Creating a Package** | # Directory structure: package\_name/ \_\_init\_\_.py module1.py module2.py | Organizes multiple modules into a package with an \_\_init\_\_.py file. | # In \_\_init\_\_.py from .module1 import function1 |
| **Import from Package** | import package\_name.module\_name | Imports a module from a package. | import mypackage.mymodule |
| **Relative Imports** | from . import module\_name | Imports a module relative to the current package. | from . import utils |
| **Access Package Modules** | package\_name.module\_name.function\_name() | Calls a function from a module within a package. | mypackage.mymodule.function1() |
| **Installing Packages** | pip install package\_name | Installs external packages from the Python Package Index. | pip install requests |
| **Checking Installed Packages** | pip list | Lists all installed packages. | pip list |
| **Uninstalling Packages** | pip uninstall package\_name | Uninstalls a package. | pip uninstall requests |
| **Package Metadata** | import pkg\_resources pkg\_resources.get\_distribution("package\_name") | Retrieves information about an installed package. | import pkg\_resources pkg\_resources.get\_distribution("numpy") |

This table provides a concise overview of key concepts and operations related to modules and packages in Python, covering how to create, use, and manage them.

4o mini

similarly give about file handling

ChatGPT

Here's a table summarizing key file handling operations and methods in Python:

| **Operation/Method** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Open a File** | open(filename, mode) | Opens a file with the specified mode. | file = open("example.txt", "r") |
| **File Modes** | - "r" (read) - "w" (write) - "a" (append) - "b" (binary) - "x" (exclusive creation) | Specifies the mode in which to open the file. | open("file.txt", "w") |
| **Read File** | file.read(size) | Reads the entire file or up to size bytes. | content = file.read() |
| **Read Line by Line** | file.readline() | Reads a single line from the file. | line = file.readline() |
| **Read All Lines** | file.readlines() | Reads all lines into a list. | lines = file.readlines() |
| **Write to File** | file.write(string) | Writes a string to the file. | file.write("Hello, World!") |
| **Append to File** | file.write(string) file.writelines(lines) | Appends text to the end of the file. | file.write("Append this line.") |
| **Close a File** | file.close() | Closes the file. | file.close() |
| **File with Context Manager** | with open(filename, mode) as file: | Opens a file and ensures it is properly closed after use. | with open("example.txt", "r") as file: content = file.read() |
| **File Seek** | file.seek(offset, whence) | Moves the file cursor to a specific position. | file.seek(0) |
| **File Tell** | file.tell() | Returns the current file cursor position. | position = file.tell() |
| **File Truncate** | file.truncate(size) | Truncates the file to the specified size. | file.truncate(100) |
| **Binary File Reading** | open(filename, "rb") | Opens a file in binary mode for reading. | file = open("example.bin", "rb") |
| **Binary File Writing** | open(filename, "wb") | Opens a file in binary mode for writing. | file = open("example.bin", "wb") |
| **Check File Existence** | os.path.exists(filename) | Checks if the file exists. | import os os.path.exists("file.txt") |
| **Delete File** | os.remove(filename) | Deletes the specified file. | os.remove("file.txt") |

This table provides a concise overview of key file handling operations and methods in Python, covering how to open, read, write, and manage files.

4o mini

similarly give about exception handling

ChatGPT

Here's a table summarizing key concepts and operations related to exception handling in Python:

| **Concept/Operation** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Try-Except Block** | try: # code except Exception as e: # code | Catches and handles exceptions raised in the try block. | try: x = 1 / 0 except ZeroDivisionError as e: print("Error:", e) |
| **Multiple Except Blocks** | try: # code except ExceptionType1 as e: # code except ExceptionType2 as e: # code | Handles different types of exceptions separately. | try: x = int("abc") except ValueError as e: print("Value Error:", e) except TypeError as e: print("Type Error:", e) |
| **Generic Exception Handling** | except: | Catches any exception if no specific exception type is provided. | try: x = int("abc") except: print("An error occurred") |
| **Finally Block** | try: # code finally: # code | Executes code after the try and except blocks, regardless of whether an exception occurred. | try: file = open("file.txt", "r") except IOError as e: print("File error:", e) finally: file.close() |
| **Raise Exception** | raise Exception("message") | Raises an exception manually. | def check\_value(x): if x < 0: raise ValueError("Negative value") |
| **Custom Exception** | class CustomError(Exception): pass | Defines a user-defined exception class. | class MyError(Exception): pass raise MyError("Something went wrong") |
| **Exception Information** | except Exception as e: print(e) | Accesses the exception instance to get details. | try: x = 1 / 0 except ZeroDivisionError as e: print("Error:", e) |
| **Assertion** | assert condition, "error message" | Tests if a condition is True; raises an AssertionError if False. | assert x > 0, "x must be positive" |

This table provides a concise overview of key exception handling features and operations in Python, covering how to handle, raise, and create exceptions.

4o mini

similarly give about oops in python

ChatGPT

Here's a table summarizing key concepts and operations related to Object-Oriented Programming (OOP) in Python:

| **Concept/Operation** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Class Definition** | class ClassName: # code | Defines a new class. | class Person: def \_\_init\_\_(self, name): self.name = name |
| **Constructor (\_\_init\_\_ method)** | def \_\_init\_\_(self, parameters): | Initializes an object when a class is instantiated. | def \_\_init\_\_(self, name): self.name = name |
| **Instance Variables** | self.variable | Variables that belong to an instance of the class. | self.name = name |
| **Instance Method** | def method\_name(self, parameters): | A method that operates on instance variables. | def greet(self): print("Hello, " + self.name) |
| **Class Method** | @classmethod def method\_name(cls, parameters): | A method that operates on the class itself. | @classmethod def from\_birth\_year(cls, name, year): return cls(name) |
| **Static Method** | @staticmethod def method\_name(parameters): | A method that does not operate on class or instance. | @staticmethod def add(x, y): return x + y |
| **Inheritance** | class SubClass(ClassName): | Inherits attributes and methods from a parent class. | class Student(Person): def \_\_init\_\_(self, name, student\_id): super().\_\_init\_\_(name) |
| **Method Overriding** | def method\_name(self, parameters): | Redefines a method in a subclass that was inherited. | def greet(self): print("Hello, student " + self.name) |
| **Encapsulation** | **Private Attribute**: self.\_\_private **Public Attribute**: self.public | Restricts access to certain attributes and methods. | self.\_\_age = 25 self.name = "John" |
| **Polymorphism** | obj.method\_name() | Methods with the same name can behave differently in different classes. | student.greet() person.greet() |
| **Abstraction** | **Abstract Class**: from abc import ABC, abstractmethod class AbstractClass(ABC): @abstractmethod def method\_name(self): pass | Defines methods that must be implemented by any subclass. | class Shape(ABC): @abstractmethod def area(self): pass |
| **Property Decorators** | @property def method\_name(self): return self.\_value | Defines getter methods that can be accessed like attributes. | @property def radius(self): return self.\_radius |

This table provides a concise overview of the core concepts and operations related to Object-Oriented Programming (OOP) in Python, covering class definitions, inheritance, encapsulation, polymorphism, and more.

4o mini

similarly give about data analysis and visualization

ChatGPT

Here's a table summarizing key concepts and operations related to data analysis and visualization in Python:

| **Concept/Operation** | **Syntax** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Pandas DataFrame Creation** | pd.DataFrame(data, columns=cols) | Creates a DataFrame from a dictionary or list. | df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]}) |
| **Read CSV File** | pd.read\_csv('filename.csv') | Reads a CSV file into a DataFrame. | df = pd.read\_csv('data.csv') |
| **Write to CSV File** | df.to\_csv('filename.csv', index=False) | Writes DataFrame to a CSV file. | df.to\_csv('data.csv', index=False) |
| **Basic Data Inspection** | df.head(n) df.tail(n) df.info() | Shows the first/last n rows and summary of DataFrame. | df.head() df.info() |
| **Descriptive Statistics** | df.describe() | Provides summary statistics for numerical columns. | df.describe() |
| **Data Selection** | df['column'] df.loc[row\_index] df.iloc[row\_index] | Selects columns or rows by label or index. | df['A'] df.loc[0] df.iloc[0] |
| **Data Filtering** | df[df['column'] > value] | Filters rows based on conditions. | df[df['A'] > 1] |
| **Group By** | df.groupby('column').agg(func) | Groups data by a column and applies aggregation functions. | df.groupby('A').sum() |
| **Sorting Data** | df.sort\_values(by='column', ascending=True) | Sorts DataFrame by one or more columns. | df.sort\_values(by='A') |
| **Handling Missing Data** | df.dropna() df.fillna(value) | Drops or fills missing values in DataFrame. | df.dropna() df.fillna(0) |
| **Plotting with Matplotlib** | plt.plot(x, y) plt.show() | Creates and displays line plots. | plt.plot(df['A'], df['B']) plt.show() |
| **Plotting with Seaborn** | sns.barplot(x='x\_col', y='y\_col', data=df) | Creates various statistical plots. | sns.barplot(x='A', y='B', data=df) |
| **Histogram** | plt.hist(data, bins=n) | Plots a histogram of data. | plt.hist(df['A'], bins=10) |
| **Scatter Plot** | plt.scatter(x, y) | Creates a scatter plot of data. | plt.scatter(df['A'], df['B']) |
| **Box Plot** | sns.boxplot(x='x\_col', y='y\_col', data=df) | Creates a box plot to show data distribution. | sns.boxplot(x='A', y='B', data=df) |
| **Line Plot** | plt.plot(x, y) | Creates a line plot of data. | plt.plot(df['A'], df['B']) |
| **Pie Chart** | plt.pie(sizes, labels=labels) | Creates a pie chart of data. | plt.pie(df['A'], labels=df['B']) |
| **Heatmap** | sns.heatmap(data) | Creates a heatmap of data values. | sns.heatmap(df.corr()) |
| **Save Plot** | plt.savefig('filename.png') | Saves the current figure to a file. | plt.savefig('plot.png') |

This table provides an overview of essential data analysis and visualization techniques in Python using libraries such as Pandas, Matplotlib, and Seaborn. It includes operations for data manipulation, statistical analysis, and creating various types of plots.

4o mini

similarly give about what is python, history of python, fetures of python , baisc installaton and setup , ides

ChatGPT

Here's a table summarizing key information about Python, including its history, features, installation, and IDEs:

| **Concept/Operation** | **Details** | **Example** |
| --- | --- | --- |
| **What is Python?** | Python is a high-level, interpreted programming language known for its readability and simplicity. It supports multiple programming paradigms including procedural, object-oriented, and functional programming. | N/A |
| **History of Python** | Python was created by Guido van Rossum and first released in 1991. It was designed with an emphasis on code readability and simplicity. Python 2 was released in 2000, and Python 3, which is the current version, was released in 2008. | N/A |
| **Features of Python** | - **Readability:** Clear and easy-to-read syntax. - **Interpreted:** Executes code line by line. - **Dynamic Typing:** No need to declare variable types. - **Object-Oriented:** Supports classes and objects. - **Extensive Standard Library:** Includes many built-in modules. - **Cross-Platform:** Runs on various operating systems. - **Community Support:** Large community and extensive documentation. | N/A |
| **Basic Installation and Setup** | 1. **Download:** Go to [python.org](https://www.python.org/) and download the installer for your operating system. 2. **Run Installer:** Execute the downloaded installer. 3. **Add to PATH:** Ensure to check the option to add Python to your system PATH. 4. **Verify Installation:** Open a command prompt or terminal and run python --version to check the installed version. | python --version |
| **IDEs** | - **IDLE:** The default Python IDE bundled with Python installation. - **PyCharm:** A popular IDE by JetBrains with extensive features for Python development. - **Visual Studio Code (VSCode):** A lightweight and powerful code editor with Python support via extensions. - **Jupyter Notebook:** An interactive environment ideal for data analysis and visualization. - **Spyder:** An IDE suited for scientific programming. | N/A |

This table provides a concise overview of Python, its history, key features, installation steps, and popular IDEs for Python development.

4o mini

is google colab also ide?

ChatGPT

Yes, Google Colab is considered an IDE, specifically an online IDE. Here’s a bit more detail about it:

| **Concept/Operation** | **Details** | **Example** |
| --- | --- | --- |
| **Google Colab** | Google Colab (Colaboratory) is a cloud-based interactive notebook environment that allows you to write and execute Python code in your browser. It is particularly popular for data analysis, machine learning, and collaborative projects. | N/A |
| **Features** | - **No Setup Required:** Runs in the cloud, so no local installation is needed. - **Interactive Notebooks:** Supports text, code, and visualizations in a notebook format. - **Integration with Google Drive:** Allows saving and sharing notebooks easily. - **Free Access to GPUs and TPUs:** Provides access to powerful hardware for computational tasks. - **Supports Python Libraries:** Comes pre-installed with many libraries for data science and machine learning. | N/A |
| **Usage** | - **Create a New Notebook:** Go to Google Colab, click on "New Notebook". - **Run Code Cells:** Write and execute Python code in cells. - **Save and Share:** Save notebooks to Google Drive and share them with others. | print("Hello, Colab!") |

Google Colab is highly valued for its ease of use, especially in educational and research contexts where sharing and collaboration are important. It provides a robust platform for executing and sharing Python code without needing any local setup.

4o mini

similarly give about Data structures introduction and DS in python

ChatGPT

Here's a table summarizing key concepts related to data structures, including an introduction and their implementation in Python:

| **Concept/Operation** | **Details** | **Example** |
| --- | --- | --- |
| **Introduction to Data Structures** | Data structures are ways to store and organize data to enable efficient access and modification. They include arrays, linked lists, stacks, queues, trees, and graphs. Choosing the right data structure can significantly affect the performance of algorithms. | N/A |
| **Lists** | Lists are ordered collections of items that can be of different types. They are mutable, meaning you can change their contents. | my\_list = [1, 2, 3, 'a', 'b'] my\_list.append(4) |
| **Tuples** | Tuples are ordered collections of items, similar to lists, but immutable. Once created, their contents cannot be changed. | my\_tuple = (1, 2, 3, 'a', 'b') element = my\_tuple[2] |
| **Sets** | Sets are unordered collections of unique items. They support mathematical operations like union, intersection, and difference. | my\_set = {1, 2, 3, 4} my\_set.add(5) |
| **Dictionaries** | Dictionaries are collections of key-value pairs. They are unordered and mutable, with keys being unique. | my\_dict = {'name': 'Alice', 'age': 25} my\_dict['age'] = 26 |
| **Stacks** | A stack is a linear data structure that follows the Last In First Out (LIFO) principle. Operations include push (add an item) and pop (remove the top item). | Implemented using a list: stack = [] stack.append(1) stack.pop() |
| **Queues** | A queue is a linear data structure that follows the First In First Out (FIFO) principle. Operations include enqueue (add an item) and dequeue (remove the front item). | Implemented using collections.deque: from collections import deque queue = deque() queue.append(1) queue.popleft() |
| **Linked Lists** | A linked list is a collection of nodes, where each node contains data and a reference to the next node. It supports efficient insertion and deletion. | Typically implemented with custom classes: class Node: def \_\_init\_\_(self, data): self.data = data self.next = None |
| **Trees** | A tree is a hierarchical data structure with a root node and child nodes. Common types include binary trees and binary search trees (BST). | Typically implemented with custom classes: class TreeNode: def \_\_init\_\_(self, value): self.value = value self.left = None self.right = None |
| **Graphs** | A graph is a collection of nodes (vertices) and edges connecting them. It can be directed or undirected, and weighted or unweighted. | Typically implemented using adjacency lists: graph = {1: [2, 3], 2: [4], 3: [4], 4: []} |
| **Arrays** | Arrays are fixed-size, homogeneous collections of elements that support efficient index-based access. Python does not have built-in arrays like other languages but provides similar functionality with lists or the array module. | import array arr = array.array('i', [1, 2, 3]) |

This table provides an overview of data structures, explaining their characteristics and how they are implemented or utilized in Python.

4o mini

similarly give about space vs time coplexity in choosing

ChatGPT

Here's a table summarizing key concepts related to space and time complexity when choosing data structures and algorithms:

| **Concept/Operation** | **Details** | **Example** |
| --- | --- | --- |
| **Time Complexity** | Measures the amount of time an algorithm takes to complete as a function of the input size. It helps evaluate the efficiency of an algorithm. Common notations include Big O (O), Big Theta (Θ), and Big Omega (Ω). | O(1): Constant time O(n): Linear time O(n^2): Quadratic time |
| **Space Complexity** | Measures the amount of memory an algorithm uses as a function of the input size. It helps assess the memory efficiency of an algorithm. | O(1): Constant space O(n): Linear space O(n^2): Quadratic space |
| **Choosing Data Structures** | The choice of data structure affects both time and space complexity. Efficient data structures can reduce the time complexity of operations and manage space better. | Use a hash table (dictionary) for fast lookups (O(1) average time complexity) |
| **Arrays** | **Time Complexity:** - Access: O(1) - Search: O(n) - Insertion/Deletion: O(n) **Space Complexity:** - O(n) (where n is the number of elements) | Efficient for access and indexing but costly for insertion and deletion. |
| **Linked Lists** | **Time Complexity:** - Access: O(n) - Search: O(n) - Insertion/Deletion: O(1) **Space Complexity:** - O(n) (additional space for pointers) | Efficient for insertions and deletions but slow for access. |
| **Stacks** | **Time Complexity:** - Push: O(1) - Pop: O(1) - Peek: O(1) **Space Complexity:** - O(n) (where n is the number of elements) | Efficient for LIFO operations with constant time complexity. |
| **Queues** | **Time Complexity:** - Enqueue: O(1) - Dequeue: O(1) - Peek: O(1) **Space Complexity:** - O(n) (where n is the number of elements) | Efficient for FIFO operations with constant time complexity. |
| **Hash Tables** | **Time Complexity:** - Search: O(1) average, O(n) worst-case - Insertion: O(1) average, O(n) worst-case - Deletion: O(1) average, O(n) worst-case **Space Complexity:** - O(n) (where n is the number of elements) | Efficient for quick lookups, insertions, and deletions. |
| **Binary Trees** | **Time Complexity:** - Search: O(log n) average for balanced trees, O(n) worst-case - Insertion/Deletion: O(log n) average for balanced trees, O(n) worst-case **Space Complexity:** - O(n) (where n is the number of nodes) | Balanced trees provide logarithmic time complexity for operations. |
| **Graphs** | **Time Complexity:** - Using adjacency list: -- Access/Update: O(1) -- Traversal (BFS/DFS): O(V + E) - Using adjacency matrix: -- Access/Update: O(1) -- Traversal (BFS/DFS): O(V^2) **Space Complexity:** - Adjacency List: O(V + E) - Adjacency Matrix: O(V^2) | Adjacency lists are space-efficient for sparse graphs, while adjacency matrices are space-efficient for dense graphs. |
| **Choosing Between Time and Space Complexity** | Often involves trade-offs. For example, choosing a hash table for fast access may use more memory, while using a simpler data structure like an array may save space but be less efficient in terms of access time. | Choosing between a hash table and a binary search tree depending on the importance of fast access vs. memory usage. |

This table provides an overview of time and space complexity concepts, helping to understand how to choose appropriate data structures based on the efficiency requirements of algorithms.

Here's a table explaining the different scopes of variables in Python:

| \*\*Scope\*\* | \*\*Description\*\* | \*\*Example\*\* |

|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|

| \*\*Local Scope\*\* | Variables defined within a function or block. They are only accessible within that function or block. | `def my\_function(): \n x = 10 \n print(x)` |

| \*\*Enclosing Scope\*\* | The scope of variables in outer functions when dealing with nested functions. Variables defined in an outer function are accessible in inner functions. | `def outer(): \n x = 10 \n def inner(): \n print(x)` |

| \*\*Global Scope\*\* | Variables defined at the top level of a script or module, outside of any function. They are accessible from anywhere in the module. | `x = 10 \n def my\_function(): \n print(x)` |

| \*\*Built-in Scope\*\* | The scope of built-in names and functions that are always available in Python. These include functions like `print()` and `len()`. | `import math \n print(math.sqrt(16))` |

### Explanation:

1. \*\*Local Scope\*\*:

- \*\*Definition\*\*: Variables created within a function or block.

- \*\*Access\*\*: Only accessible within that function or block.

- \*\*Lifetime\*\*: Exists only while the function or block is executing.

- \*\*Example\*\*:

```python

def my\_function():

x = 10 # Local variable

print(x)

my\_function() # Outputs: 10

# print(x) # This would raise an error because x is not accessible outside the function.

```

2. \*\*Enclosing Scope\*\*:

- \*\*Definition\*\*: The scope of variables in outer functions when there are nested functions.

- \*\*Access\*\*: Inner functions can access variables from their enclosing (outer) functions.

- \*\*Lifetime\*\*: Exists as long as the enclosing function is still in scope.

- \*\*Example\*\*:

```python

def outer():

x = 10 # Enclosing variable

def inner():

print(x) # Accesses variable x from outer

inner()

outer() # Outputs: 10

```

3. \*\*Global Scope\*\*:

- \*\*Definition\*\*: Variables defined at the top level of a script or module.

- \*\*Access\*\*: Accessible from any part of the module or script.

- \*\*Lifetime\*\*: Exists for the entire runtime of the module or script.

- \*\*Example\*\*:

```python

x = 10 # Global variable

def my\_function():

print(x) # Accesses global variable x

my\_function() # Outputs: 10

print(x) # Outputs: 10

```

4. \*\*Built-in Scope\*\*:

- \*\*Definition\*\*: The scope of built-in names and functions that are always available in Python.

- \*\*Access\*\*: Accessible from anywhere in the code without needing to import or define them.

- \*\*Lifetime\*\*: Always available as long as Python is running.

- \*\*Example\*\*:

```python

print(len("Hello")) # Built-in function len()

import math

print(math.sqrt(16)) # Built-in module function sqrt()

```

### Scope Resolution

- \*\*LEGB Rule\*\*: Python follows the LEGB rule to resolve variable names, where it looks up variables in the following order: Local, Enclosing, Global, Built-in.

Understanding these scopes helps in managing variable accessibility and avoiding potential name conflicts in your code.

To cover the broad range of Python concepts effectively, here are three important programs that collectively demonstrate various key aspects:

### Program 1: \*\*Python Basics and Data Structures\*\*

This program introduces basic Python syntax, variables, data types, operators, control flow, functions, and basic data structures.

```python

# Basic Python program demonstrating syntax, variables, data types, operators, and control flow

# Variables and Data Types

name = "Alice" # String

age = 30 # Integer

height = 5.6 # Float

is\_student = True # Boolean

# Print statements

print("Name:", name)

print("Age:", age)

print("Height:", height)

print("Is Student:", is\_student)

# Operators

sum\_age = age + 5

is\_adult = age >= 18

print("Sum of Age and 5:", sum\_age)

print("Is Adult:", is\_adult)

# Control Flow

if is\_adult:

print(f"{name} is an adult.")

else:

print(f"{name} is not an adult.")

# Functions

def greet(person\_name):

return f"Hello, {person\_name}!"

print(greet(name))

# Data Structures

# List

fruits = ["apple", "banana", "cherry"]

print("Fruits:", fruits)

# Dictionary

person\_info = {"name": name, "age": age, "height": height}

print("Person Info:", person\_info)

# Loop

for fruit in fruits:

print(fruit)

# Example of exception handling

try:

result = 10 / 0

except ZeroDivisionError:

print("Cannot divide by zero!")

```

\*\*Concepts Covered:\*\*

- \*\*Basic Syntax:\*\* Writing and running a Python program.

- \*\*Variables and Data Types:\*\* String, integer, float, boolean.

- \*\*Operators:\*\* Arithmetic and comparison.

- \*\*Control Flow:\*\* `if` statements.

- \*\*Functions:\*\* Defining and using functions.

- \*\*Data Structures:\*\* Lists, dictionaries.

- \*\*Exception Handling:\*\* Basic error handling.

### Program 2: \*\*File Handling, Modules, and Packages\*\*

This program demonstrates how to handle files, use modules, and include basic package functionality.

```python

# Importing modules

import os

# File Handling

def write\_to\_file(filename, content):

try:

with open(filename, 'w') as file:

file.write(content)

print("File written successfully.")

except Exception as e:

print(f"An error occurred: {e}")

def read\_from\_file(filename):

try:

with open(filename, 'r') as file:

return file.read()

except FileNotFoundError:

return "File not found."

except Exception as e:

return f"An error occurred: {e}"

# Example usage

filename = 'example.txt'

write\_to\_file(filename, "Hello, Python World!")

print("File Content:", read\_from\_file(filename))

# Using the os module

print("Current Working Directory:", os.getcwd())

# Creating a simple module

def add(x, y):

return x + y

# Save the module in a separate file named 'mymodule.py'

# Code in 'mymodule.py':

# def add(x, y):

# return x + y

# Importing the custom module

import mymodule

print("Addition using custom module:", mymodule.add(5, 3))

```

\*\*Concepts Covered:\*\*

- \*\*File Handling:\*\* Reading from and writing to files.

- \*\*Modules:\*\* Importing and using standard and custom modules.

- \*\*Packages:\*\* Usage and basic package creation.

### Program 3: \*\*OOP, Basic Data Analysis, and Visualization\*\*

This program introduces object-oriented programming (OOP), performs basic data analysis, and visualizes data using `matplotlib`.

```python

import matplotlib.pyplot as plt

import statistics

# Object-Oriented Programming (OOP)

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

return f"Hello, my name is {self.name} and I am {self.age} years old."

# Example usage

person = Person("Alice", 30)

print(person.greet())

# Basic Data Analysis

data = [10, 20, 30, 40, 50]

mean = statistics.mean(data)

median = statistics.median(data)

stdev = statistics.stdev(data)

print("Data:", data)

print("Mean:", mean)

print("Median:", median)

print("Standard Deviation:", stdev)

# Basic Visualization

plt.figure(figsize=(8, 5))

plt.plot(data, marker='o')

plt.title('Basic Data Visualization')

plt.xlabel('Index')

plt.ylabel('Value')

plt.grid(True)

plt.show()

```

\*\*Concepts Covered:\*\*

- \*\*OOP:\*\* Classes and objects, methods.

- \*\*Basic Data Analysis:\*\* Mean, median, standard deviation.

- \*\*Basic Data Visualization:\*\* Using `matplotlib` for plotting data.

### Summary of Concepts Covered

1. \*\*Python Basics and Data Structures:\*\* Syntax, variables, data types, operators, control flow, functions, basic data structures.

2. \*\*File Handling, Modules, and Packages:\*\* File operations, using standard and custom modules, package usage.

3. \*\*OOP, Basic Data Analysis, and Visualization:\*\* Object-oriented programming, data analysis with statistics, data visualization using `matplotlib`.

These programs provide a comprehensive overview of fundamental Python concepts and practical applications, helping students understand the essentials of Python programming and its various features.

* [Introduction](https://cs50.harvard.edu/web/2020/notes/2/#introduction)
* [Python](https://cs50.harvard.edu/web/2020/notes/2/#python)
* [Variables](https://cs50.harvard.edu/web/2020/notes/2/#variables)
* [Formatting Strings](https://cs50.harvard.edu/web/2020/notes/2/#formatting-strings)
* [Conditions](https://cs50.harvard.edu/web/2020/notes/2/#conditions)
* [Sequences](https://cs50.harvard.edu/web/2020/notes/2/#sequences)
  + [Strings](https://cs50.harvard.edu/web/2020/notes/2/#strings)
  + [Lists](https://cs50.harvard.edu/web/2020/notes/2/#lists)
  + [Tuples](https://cs50.harvard.edu/web/2020/notes/2/#tuples)
  + [Sets](https://cs50.harvard.edu/web/2020/notes/2/#sets)
  + [Dictionaries](https://cs50.harvard.edu/web/2020/notes/2/#dictionaries)
  + [Loops](https://cs50.harvard.edu/web/2020/notes/2/#loops)
* [Functions](https://cs50.harvard.edu/web/2020/notes/2/#functions)
* [Modules](https://cs50.harvard.edu/web/2020/notes/2/#modules)
* [Object-Oriented Programming](https://cs50.harvard.edu/web/2020/notes/2/#object-oriented-programming)
* [Functional Programming](https://cs50.harvard.edu/web/2020/notes/2/#functional-programming)
  + [Decorators](https://cs50.harvard.edu/web/2020/notes/2/#decorators)
  + [Lambda Functions](https://cs50.harvard.edu/web/2020/notes/2/#lambda-functions)
* [Exceptions](https://cs50.harvard.edu/web/2020/notes/2/#exceptions)

## [Introduction](https://cs50.harvard.edu/web/2020/notes/2/#introduction)

* So far, we’ve discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others.
* Today, we’ll dive into Python, one of the two main programming languages we’ll use throughout this course.

## [Python](https://cs50.harvard.edu/web/2020/notes/2/#python)



* Python is a very powerful and widely-used language that will allow us to quickly build fairly complicated web applications. In this course, we’ll be using Python 3, although Python 2 is still in use in some places. When looking at outside resources, be careful to make sure they’re using the same version.
* Let’s start where we start with many programming languages: Hello, world. This program, written in Python, would look like this:

print("Hello, world!")

* To break down what’s going on in that line, there is a print **function** built in to the python language, that takes an **argument** in parentheses, and displays that argument on the command line.
* To actually write and run this program on your computers, you’ll first type this line into your text editor of choice, and then save the file as something.py. Next, you’ll head over to your terminal, navigate to the directory containing your file, and type python something.py. In the case of the above program, the words “Hello, world!” will then be displayed in the terminal.
* Depending on how your computer is set up, you may have to type python3 instead of python before the file name, and you may even have to [download Python](https://www.python.org/downloads/) if you haven’t already. After installing Python, we recommend that you also [download Pip](https://pip.pypa.io/en/stable/installing/), as you’ll need that later in the course.
* When you type python file.py in your terminal, a program called an **interpreter**, which you downloaded together with Python, reads through your file line by line, and executes each line of the code. This is different than languages like **C** or **Java**, which need to be **compiled** into machine code before they can be run.

## [Variables](https://cs50.harvard.edu/web/2020/notes/2/#variables)

A key part of any programming language is the ability to create and manipulate variables. In order to assign a value to a variable in Python, the syntax looks like this:

a = 28

b = 1.5

c = "Hello!"

d = True

e = None

Each of these lines is taking the value to the right of the =, and storing it in the variable name to the left.

Unlike in some other programming languages, Python variable types are inferred, meaning that while each variable does have a type, we do not have to explicitly state which type it is when we create the variable. Some of the most common variable types are:

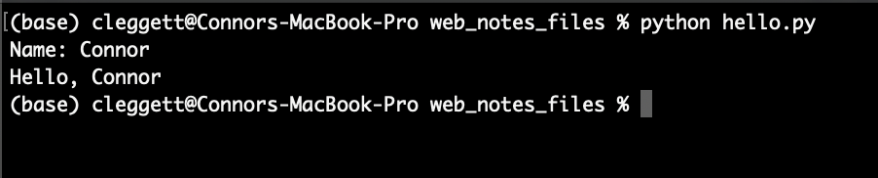
* **int**: An integer
* **float**: A decimal number
* **str**: A string, or sequence of characters
* **bool**: A value that is either True or False
* **NoneType**: A special value (None) indicating the absence of a value.

Now, we’ll work on writing a more interesting program that can take input from the user and say hello to that user. To do this, we’ll use another built in function called input which displays a prompt to the user, and returns whatever the user provides as input. For example, we can write the following in a file called name.py:

name = input("Name: ")

print("Hello, " + name)

When run on the terminal, this is what the program looks like:



A couple of things to point out here:

* In the first line, instead of assigning the variable name to an explicit value, we’re assigning it to whatever the input function returns.
* In the second line, we’re using the + operator to combine, or **concatenate**, two strings. In python, the + operator can be used to add numbers or concatenate strings and lists.

## [Formatting Strings](https://cs50.harvard.edu/web/2020/notes/2/#formatting-strings)

* While we can use the + operator to combine strings as we did above, in the latest versions of python, there are even easier ways to work with strings, known as [formatted strings](https://realpython.com/python-f-strings/), or **f-strings** for short.
* To indicate that we’re using formatted strings, we simply add an f before the quotation marks. For example, instead of using "Hello, " + name as we did above, we could write f"Hello, {name}" for the same result. We can even plug a function into this string if we want, and turn our program above into the single line:

print(f"Hello, **{**input("Name**:** ")**}**")

## [Conditions](https://cs50.harvard.edu/web/2020/notes/2/#conditions)

* Just like in other programming languages, Python gives us the ability to run different segments of code based on different [conditions](https://realpython.com/python-conditional-statements/). For example, in the program below, we’ll change our output depending on the number a user types in:

num = input("Number: ")

**if** num > 0:

print("Number is positive")

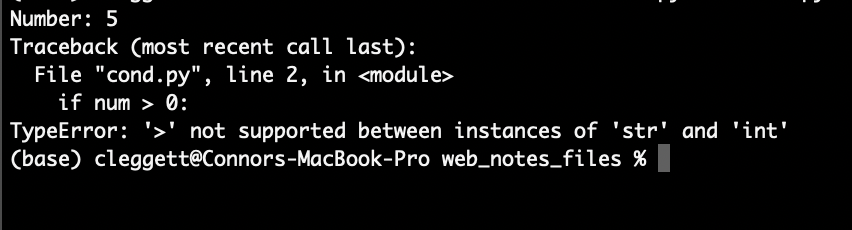
**elif** num < 0:

print("Number is negative")

**else**:

print("Number is 0")

* Getting into how the above program works, conditionals in python contain a keyword (if, elif, or else) and then (except in the else case) a boolean expression, or an expression that evaluates to either True or False. Then, all of the code we want to run if a certain expression is true is **indented** directly below the statement. Indentation is required as part of the Python syntax.
* However, when we run this program, we run into an [exception](https://docs.python.org/3/tutorial/errors.html) that looks like this:



* An exception is what happens when an error occurs while we’re running our python code, and over time you’ll get better and better at interpreting these errors, which is a very valuable skill to have.
* Let’s look a bit more closely at this specific exception: If we look at the bottom, we’ll see that we ran into a TypeError, which generally means Python expected a certain variable to be of one type, but found it to be of another type. In this case, the exception tells us that we cannot use the > symbol to compare a str and int, and then above we can see that this comparison occurs in line 2.
* In this case, it’s obvious that 0 is an integer, so it must be the case that our num variable is a string. This is happening because it turns out that the input function always returns a string, and we have to specify that it should be turned into (or **cast** into) an integer using the int function. This means our first line would now look like:

num = int(input("Number: "))

* Now, the program will work just as we intended!

## [Sequences](https://cs50.harvard.edu/web/2020/notes/2/#sequences)

One of the most powerful parts of the Python language is its ability to work with **sequences** of data in addition to individual variables.

There are several types of sequences that are similar in some ways, but different in others. When explaining those differences, we’ll use the terms **mutable/immutable** and **ordered/unordered**. **Mutable** means that once a sequence has been defined, we can change individual elements of that sequence, and **ordered** means that the order of the objects matters.

### [Strings](https://cs50.harvard.edu/web/2020/notes/2/#strings)

**Ordered**: Yes

**Mutable**: No

We’ve already looked at strings a little bit, but instead of just variables, we can think of a string as a sequence of characters. This means we can access individual elements within the string! For example:

name = "Harry"

print(name[0])

print(name[1])

prints out the first (or index-0) character in the string, which in this case happens to be H, and then prints out the second (or index-1) character, which is a.

### [Lists](https://cs50.harvard.edu/web/2020/notes/2/#lists)

**Ordered**: Yes

**Mutable**: Yes

A [Python list](https://www.w3schools.com/python/python_lists.asp) allows you to store any variable types. We create a list using square brackets and commas, as shown below. Similarly to strings, we can print an entire list, or some individual elements. We can also add elements to a list using append, and sort a list using sort

*# This is a Python comment*

names = ["Harry", "Ron", "Hermione"]

*# Print the entire list:*

print(names)

*# Print the second element of the list:*

print(names[1])

*# Add a new name to the list:*

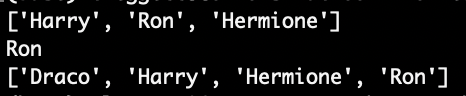
names.append("Draco")

*# Sort the list:*

names.sort()

*# Print the new list:*

print(names)



### [Tuples](https://cs50.harvard.edu/web/2020/notes/2/#tuples)

**Ordered**: Yes

**Mutable**: No

[Tuples](https://www.w3schools.com/python/python_tuples.asp) are generally used when you need to store just two or three values together, such as the x and y values for a point. In Python code, we use parentheses:

point = (12.5, 10.6)

### [Sets](https://cs50.harvard.edu/web/2020/notes/2/#sets)

**Ordered**: No

**Mutable**: N/A

[Sets](https://www.w3schools.com/python/python_sets.asp) are different from lists and tuples in that they are **unordered**. They are also different because while you can have two or more of the same elements within a list/tuple, a set will only store each value once. We can define an empty set using the set function. We can then use add and remove to add and remove elements from that set, and the len function to find the set’s size. Note that the len function works on all sequences in python. Also note that despite adding 4 and 3 to the set twice, each item can only appear once in a set.

*# Create an empty set:*

s = set()

*# Add some elements:*

s.add(1)

s.add(2)

s.add(3)

s.add(4)

s.add(3)

s.add(1)

*# Remove 2 from the set*

s.remove(2)

*# Print the set:*

print(s)

*# Find the size of the set:*

print(f"The set has **{**len(s)**}** elements.")

""" This is a python multi-line comment:

Output:

{1, 3, 4}

The set has 3 elements.

"""

### [Dictionaries](https://cs50.harvard.edu/web/2020/notes/2/#dictionaries)

**Ordered**: No

**Mutable**: Yes

[Python Dictionaries](https://www.w3schools.com/python/python_dictionaries.asp) or dicts, will be especially useful in this course. A dictionary is a set of **key-value pairs**, where each key has a corresponding value, just like in a dictionary, each word (the key) has a corresponding definition (the value). In Python, we use curly brackets to contain a dictionary, and colons to indicate keys and values. For example:

*# Define a dictionary*

houses = {"Harry": "Gryffindor", "Draco": "Slytherin"}

*# Print out Harry's house*

print(houses["Harry"])

*# Adding values to a dictionary:*

houses["Hermione"] = "Gryffindor"

*# Print out Hermione's House:*

print(houses["Hermione"])

""" Output:

Gryffindor

Gryffindor

"""

### [Loops](https://cs50.harvard.edu/web/2020/notes/2/#loops)

Loops are an incredibly important part of any programming language, and in Python, they come in two main forms: [for loops](https://www.w3schools.com/python/python_for_loops.asp) and [while loops](https://www.w3schools.com/python/python_while_loops.asp). For now, we’ll focus on For Loops.

* For loops are used to iterate over a sequence of elements, performing some block of code (indented below) for each element in a sequence. For example, the following code will print out the numbers from 0 to 5:

**for** i **in** [0, 1, 2, 3, 4, 5]:

print(i)

""" Output:

0

1

2

3

4

5

"""

* We can condense this code using the python range function, which allows us to easily get a sequence of numbers. The following code gives the exact same result as our code from above:

**for** i **in** range(6):

print(i)

""" Output:

0

1

2

3

4

5

"""

* This type of loop can work for any sequence! For example, if we wish to print each name in a list, we could write the code below:

*# Create a list:*

names = ["Harry", "Ron", "Hermione"]

*# Print each name:*

**for** name **in** names:

print(name)

""" Output:

Harry

Ron

Hermione

"""

* We can get even more specific if we want, and loop through each character in a single name!

name = "Harry"

**for** char **in** name:

print(char)

""" Output:

H

a

r

r

y

"""

## [Functions](https://cs50.harvard.edu/web/2020/notes/2/#functions)

We’ve already seen a few python functions such as print and input, but now we’re going to dive into writing our own functions. To get started, we’ll write a function that takes in a number and squares it:

**def** square(x):

**return** x \* x

Notice how we use the def keyword to indicate we’re defining a function, that we’re taking in a single input called x and that we use the return keyword to indicate what the function’s output should be.

We can then “call” this function just as we’ve called other ones: using parentheses:

**for** i **in** range(10):

print(f"The square of **{**i**}** is **{**square(i)**}**")

""" Output:

The square of 0 is 0

The square of 1 is 1

The square of 2 is 4

The square of 3 is 9

The square of 4 is 16

The square of 5 is 25

The square of 6 is 36

The square of 7 is 49

The square of 8 is 64

The square of 9 is 81

"""

## [Modules](https://cs50.harvard.edu/web/2020/notes/2/#modules)

As our projects get larger and larger, it will become useful to be able to write functions in one file and run them in another. In the case above, we could create create one file called functions.py with the code:

**def** square(x):

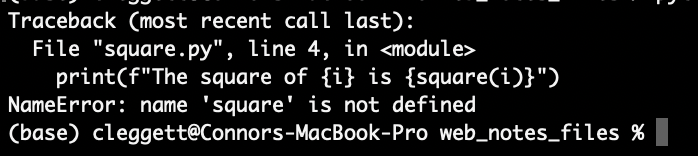
**return** x \* x

And another file called square.py with the code:

**for** i **in** range(10):

print(f"The square of **{**i**}** is **{**square(i)**}**")

However, when we try to run square.py, we run into the following error:



We run into this problem because by default, Python files don’t know about each other, so we have to explicitly import the square function from the functions **module** we just wrote. Now, when square.py looks like this:

**from** functions **import** square

**for** i **in** range(10):

print(f"The square of **{**i**}** is **{**square(i)**}**")

Alternatively, we can choose to import the entire functions module and then use dot notation to access the square function:

**import** functions

**for** i **in** range(10):

print(f"The square of **{**i**}** is **{**functions.square(i)**}**")

There are many built-in Python modules we can import such as math or csv that give us access to even more functions. Additionally, we can download even more Modules to access even more functionality! We’ll spend a lot of time using the Django Module, which we’ll discuss in the next lecture.

## [Object-Oriented Programming](https://cs50.harvard.edu/web/2020/notes/2/#object-oriented-programming)

[Object Oriented Programming](https://en.wikipedia.org/wiki/Object-oriented_programming) is a programming paradigm, or a way of thinking about programming, that is centered around objects that can store information and perform actions.

* **Classes**: We’ve already seen a few different types of variables in python, but what if we want to create our own type? A [Python Class](https://www.w3schools.com/python/python_classes.asp) is essentially a template for a new type of object that can store information and perform actions. Here’s a class that defines a two-dimensional point:

**class** **Point**():

*# A method defining how to create a point:*

**def** \_\_init\_\_(self, x, y):

self.x = x

self.y = y

* Note that in the above code, we use the keyword self to represent the object we are currently working with. self should be the first argument for any method within a Python class.

Now, let’s see how we can actually use the class from above to create an object:

p = **Point**(2, 8)

print(p.x)

print(p.y)

""" Output:

2

8

"""

Now, let’s look at a more interesting example where instead of storing just the coordinates of a Point, we create a class that represents an airline flight:

**class** **Flight**():

*# Method to create new flight with given capacity*

**def** \_\_init\_\_(self, capacity):

self.capacity = capacity

self.passengers = []

*# Method to add a passenger to the flight:*

**def** add\_passenger(self, name):

self.passengers.append(name)

However, this class is flawed because while we set a capacity, we could still add too many passengers. Let’s augment it so that before adding a passenger, we check to see if there is room on the flight:

**class** **Flight**():

*# Method to create new flight with given capacity*

**def** \_\_init\_\_(self, capacity):

self.capacity = capacity

self.passengers = []

*# Method to add a passenger to the flight:*

**def** add\_passenger(self, name):

**if** **not** self.open\_seats():

**return** False

self.passengers.append(name)

**return** True

*# Method to return number of open seats*

**def** open\_seats(self):

**return** self.capacity - len(self.passengers)

Note that above, we use the line if not self.open\_seats() to determine whether or not there are open seats. This works because in Python, the number 0 can be interpretted as meaning False, and we can also use the keyword not to signify the opposite of the following statement so not True is False and not False is True. Therefore, if open\_seats returns 0, the entire expression will evaluate to True

Now, let’s try out the class we’ve created by instantiating some objects:

*# Create a new flight with o=up to 3 passengers*

flight = **Flight**(3)

*# Create a list of people*

people = ["Harry", "Ron", "Hermione", "Ginny"]

*# Attempt to add each person in the list to a flight*

**for** person **in** people:

**if** flight.add\_passenger(person):

print(f"Added **{**person**}** to flight successfully")

**else**:

print(f"No available seats for **{**person**}**")

""" Output:

Added Harry to flight successfully

Added Ron to flight successfully

Added Hermione to flight successfully

No available seats for Ginny

"""

## [Functional Programming](https://cs50.harvard.edu/web/2020/notes/2/#functional-programming)

In addition to supporting Object-Oriented Programming, Python also supports the [Functional Programming Paradigm](https://en.wikipedia.org/wiki/Functional_programming), in which functions are treated as values just like any other variable.

### [Decorators](https://cs50.harvard.edu/web/2020/notes/2/#decorators)

One thing made possible by functional programming is the idea of a decorator, which is a higher-order function that can modify another function. For example, let’s write a decorator that announces when a function is about to begin, and when it ends. We can then apply this decorator using an @ symbol.

**def** announce(f):

**def** wrapper():

print("About to run the function")

f()

print("Done with the function")

**return** wrapper

@announce

**def** hello():

print("Hello, world!")

hello()

""" Output:

About to run the function

Hello, world!

Done with the function

"""

### [Lambda Functions](https://cs50.harvard.edu/web/2020/notes/2/#lambda-functions)

Lambda functions provide another way to create functions in python. For example, if we want to define the same square function we did earlier, we can write:

square = **lambda** x: x \* x

Where the input is to the left of the : and the output is on the right.

This can be useful when we don’t want to write a whole separate function for a single, small use. For example, if we want to sort some objects where it’s not clear at first how to sort them. Imagine we have a list of people, but with names and houses instead of just names that we wish to sort:

people = [

{"name": "Harry", "house": "Gryffindor"},

{"name": "Cho", "house": "Ravenclaw"},

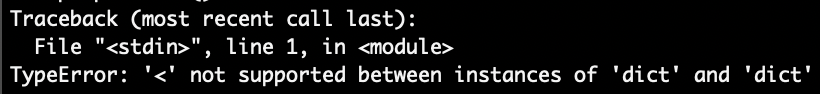
{"name": "Draco", "house": "Slytherin"}

]

people.sort()

print(people)

This, however, leaves us with the error:



Which occurs because Python doesn’t know how to compare two Dictionaries to check if one is less than the other.

We can solve this problem by including a key argument to the sort function, which specifies which part of the dictionary we wish to use to sort:

people = [

{"name": "Harry", "house": "Gryffindor"},

{"name": "Cho", "house": "Ravenclaw"},

{"name": "Draco", "house": "Slytherin"}

]

**def** f(person):

**return** person["name"]

people.sort(key=f)

print(people)

""" Output:

[{'name': 'Cho', 'house': 'Ravenclaw'}, {'name': 'Draco', 'house': 'Slytherin'}, {'name': 'Harry', 'house': 'Gryffindor'}]

"""

While this does work, we’ve had to write an entire function that we’re only using once, we can make our code more readable by using a lambda function:

people = [

{"name": "Harry", "house": "Gryffindor"},

{"name": "Cho", "house": "Ravenclaw"},

{"name": "Draco", "house": "Slytherin"}

]

people.sort(key=**lambda** person: person["name"])

print(people)

""" Output:

[{'name': 'Cho', 'house': 'Ravenclaw'}, {'name': 'Draco', 'house': 'Slytherin'}, {'name': 'Harry', 'house': 'Gryffindor'}]

"""

## [Exceptions](https://cs50.harvard.edu/web/2020/notes/2/#exceptions)

During this lecture, we’ve run into a few different exceptions, so now we’ll look into some new ways of dealing with them.

In the following chunk of code, we’ll take two integers from the user, and attempt to divide them:

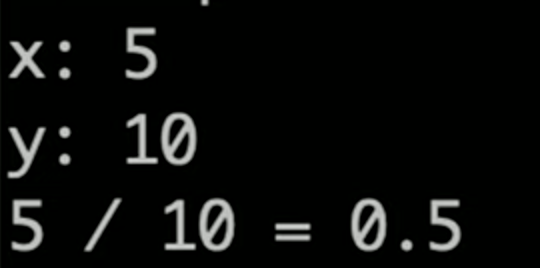
x = int(input("x: "))

y = int(input("y: "))

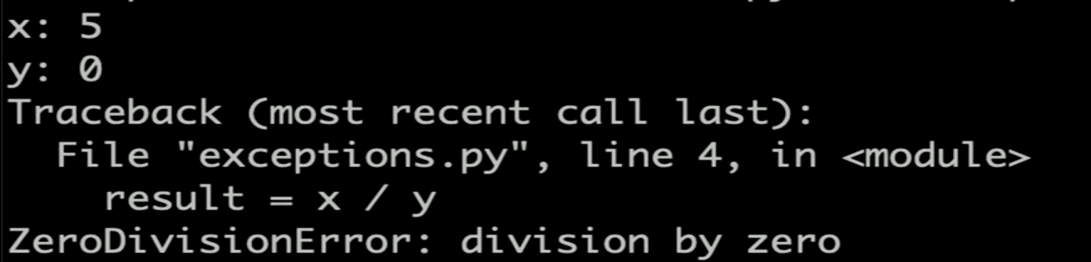
result = x / y

print(f"**{**x**}** / **{**y**}** = **{**result**}**")

In many cases, this program works well:



However, we’ll run into problems when we attempt to divide by 0:



We can deal with this messy error using [Exception Handling](https://www.w3schools.com/python/python_try_except.asp). In the following block of code, we will try to divide the two numbers, except when we get a ZeroDivisionError:

**import** sys

x = int(input("x: "))

y = int(input("y: "))

**try**:

result = x / y

**except** ZeroDivisionError:

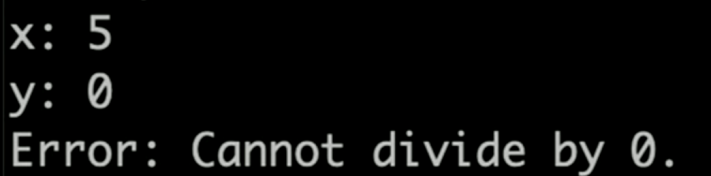
print("Error: Cannot divide by 0.")

*# Exit the program*

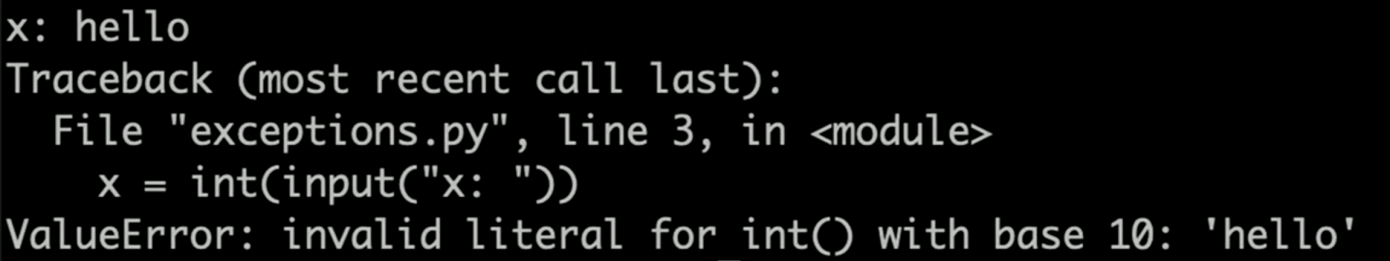
sys.exit(1)

print(f"**{**x**}** / **{**y**}** = **{**result**}**")

In this case, when we try it again:



However, we still run into an error when the user enters non-numbers for x and y:



We can solve this problem in a similar manner!

**import** sys

**try**:

x = int(input("x: "))

y = int(input("y: "))

**except** ValueError:

print("Error: Invalid input")

sys.exit(1)

**try**:

result = x / y

**except** ZeroDivisionError:

print("Error: Cannot divide by 0.")

*# Exit the program*

sys.exit(1)

print(f"**{**x**}** / **{**y**}** = **{**result**}**")

That’s all for this lecture! Next time, we’ll use Python’s Django Module to build some applications!