Sure! Here's a concise yet detailed study guide on **Python fundamentals** focused on **dictionaries, tuples, functions, and looping constructs**:

**1. Python Fundamentals: Dictionaries, Tuples, Functions, and Looping Constructs**

**1. Dictionaries**

* **Definition:** An unordered, mutable collection of key-value pairs.
* **Syntax:**
* my\_dict = {'key1': 'value1', 'key2': 'value2'}
* **Key points:**
  + Keys must be immutable types (strings, numbers, tuples).
  + Values can be any Python data type.
  + Access values by keys: my\_dict['key1']
  + Add/update: my\_dict['key3'] = 'value3'
  + Remove: del my\_dict['key2']
  + Useful methods:
    - my\_dict.keys() — returns keys
    - my\_dict.values() — returns values
    - my\_dict.items() — returns key-value pairs

**2. Tuples**

* **Definition:** An ordered, immutable collection of items.
* **Syntax:**
* my\_tuple = (1, 2, 3)
* **Key points:**
  + Tuples cannot be changed after creation.
  + Useful for fixed collections and as dictionary keys.
  + Can be unpacked: a, b, c = my\_tuple
  + Single element tuple: (5,) (comma is necessary)

**3. Functions**

* **Definition:** Blocks of reusable code that perform a specific task.
* **Syntax:**
* def function\_name(parameters):
* # code block
* return value # optional
* **Key points:**
  + Parameters can have default values.
  + Functions can return values or nothing (None).
  + Can have \*args and \*\*kwargs for variable-length arguments.
  + Example:
  + def greet(name="User"):
  + print(f"Hello, {name}!")

**4. Looping Constructs**

**a) for loop**

* Used to iterate over a sequence (list, tuple, dictionary, string).
* Syntax:
* for item in iterable:
* # code block
* Example:
* for i in range(5):
* print(i)

**b) while loop**

* Repeats as long as a condition is True.
* Syntax:
* while condition:
* # code block
* Example:
* count = 0
* while count < 5:
* print(count)
* count += 1

**c) Loop Control Statements**

* break — exit loop immediately
* continue — skip current iteration
* pass — do nothing (placeholder)

**Example Integrating All Concepts**

def count\_values(d):

# Count how many values are greater than 10

count = 0

for key, value in d.items():

if value > 10:

count += 1

return count

my\_dict = {'a': 5, 'b': 15, 'c': 25}

print(count\_values(my\_dict)) # Output: 2

If you want, I can provide exercises or code examples for practice!

Absolutely! Here’s a detailed study guide on **Object-Oriented Programming (OOP) concepts, generators, file I/O, and regular expressions in Python**:

**2. Python Advanced Concepts: OOP, Generators, File I/O, and Regular Expressions**

**1. Object-Oriented Programming (OOP) Concepts**

**Key Principles:**

* **Class**: A blueprint to create objects (instances) with attributes (variables) and methods (functions).
* class Car:
* def \_\_init\_\_(self, make, model):
* self.make = make # attribute
* self.model = model
* def display(self):
* print(f"Car: {self.make} {self.model}")
* **Object**: An instance of a class.
* my\_car = Car('Toyota', 'Corolla')
* my\_car.display()
* **Encapsulation**: Bundling data (attributes) and methods that operate on the data inside one unit (class). Control access using private/protected attributes (\_var, \_\_var).
* **Inheritance**: Creating a new class from an existing class to reuse or extend functionality.
* class ElectricCar(Car):
* def \_\_init\_\_(self, make, model, battery\_size):
* super().\_\_init\_\_(make, model)
* self.battery\_size = battery\_size
* **Polymorphism**: Methods with the same name behave differently in different classes.
* class Dog:
* def speak(self):
* print("Woof!")
* class Cat:
* def speak(self):
* print("Meow!")
* def animal\_speak(animal):
* animal.speak() # works differently based on animal type
* **Abstraction**: Hiding complex implementation details and showing only the necessary features.

**2. Generators**

* **Definition**: Functions that return an iterator and produce items one at a time using the yield keyword.
* **Why use generators?**
  + Memory efficient for large datasets.
  + Generate values on the fly (lazy evaluation).
* **Syntax:**
* def count\_up\_to(n):
* count = 1
* while count <= n:
* yield count
* count += 1
* **Usage:**
* for number in count\_up\_to(5):
* print(number)
* Generators can be iterated once and maintain their state between calls.

**3. File I/O (Input/Output)**

* **Opening a file:**
* f = open('file.txt', 'r') # 'r' for read mode
* **Modes:**
  + 'r': read (default)
  + 'w': write (overwrite)
  + 'a': append
  + 'r+': read and write
  + 'b': binary mode (append to mode, e.g. 'rb')
* **Reading files:**
* contents = f.read() # read entire file
* line = f.readline() # read one line
* lines = f.readlines() # read all lines into list
* **Writing files:**
* f = open('output.txt', 'w')
* f.write("Hello, world!\n")
* f.close()
* **Best practice: Use with statement to auto-close files**
* with open('file.txt', 'r') as f:
* for line in f:
* print(line.strip())

**4. Regular Expressions (Regex)**

* **Purpose**: Pattern matching and text manipulation.
* **Python module:** import re
* **Common functions:**
  + re.match(pattern, string): matches at start of string
  + re.search(pattern, string): search anywhere in string
  + re.findall(pattern, string): list of all matches
  + re.sub(pattern, repl, string): replace occurrences
* **Basic syntax:**
  + . — any character except newline
  + \* — 0 or more repetitions
  + + — 1 or more repetitions
  + ? — 0 or 1 repetition
  + \d — digit [0-9]
  + \w — word character [a-zA-Z0-9\_]
  + ^ — start of string
  + $ — end of string
  + [abc] — match any one of a, b, or c
  + {m,n} — between m and n repetitions
* **Example:**
* import re
* text = "My phone number is 123-456-7890."
* pattern = r'\d{3}-\d{3}-\d{4}'
* match = re.search(pattern, text)
* if match:
* print("Phone number found:", match.group())

If you want, I can provide sample code snippets or practice exercises for any of these topics!

Sure! Here's a detailed study guide on **how to create and import various modules and packages in Python**:

**3. Creating and Importing Modules and Packages in Python**

**1. What is a Module?**

* A **module** is a single Python file (.py) that contains Python code — functions, classes, variables, or runnable code.
* Modules help organize code into reusable, maintainable components.
* Example: math.py, os.py are built-in modules.

**2. Creating a Module**

* To create a module, simply write Python code in a .py file.

Example: Create a file named mymath.py with:

# mymath.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

PI = 3.14159

**3. Importing Modules**

You can use modules by importing them in your script or interactive session.

**Ways to import:**

* **Import the whole module**

import mymath

print(mymath.add(5, 3)) # Output: 8

print(mymath.PI) # Output: 3.14159

* **Import specific functions or variables**

from mymath import add, PI

print(add(2, 7)) # Output: 9

print(PI) # Output: 3.14159

* **Import with alias**

import mymath as mm

print(mm.subtract(10, 4)) # Output: 6

**4. What is a Package?**

* A **package** is a directory (folder) that contains multiple modules.
* It must contain a special file called \_\_init\_\_.py (can be empty) to be recognized as a package by Python.
* Packages allow for hierarchical structuring of modules.

Example:

mypackage/

\_\_init\_\_.py

module1.py

module2.py

**5. Creating a Package**

* Create a directory named mypackage.
* Inside that directory, create \_\_init\_\_.py (can be empty or contain initialization code).
* Add Python files (modules) like module1.py, module2.py inside.

Example:

mypackage/module1.py

def greet():

print("Hello from module1")

mypackage/module2.py

def welcome():

print("Welcome from module2")

**6. Importing from a Package**

* **Import module from package**

import mypackage.module1

mypackage.module1.greet() # Output: Hello from module1

* **Import specific function**

from mypackage.module2 import welcome

welcome() # Output: Welcome from module2

* **Import package modules in \_\_init\_\_.py**

You can expose specific modules or functions by importing them inside mypackage/\_\_init\_\_.py.

Example mypackage/\_\_init\_\_.py:

from .module1 import greet

from .module2 import welcome

Now you can do:

import mypackage

mypackage.greet() # Hello from module1

mypackage.welcome() # Welcome from module2

**7. Using Modules and Packages from Other Directories**

* Ensure the package/module directory is in the **Python Path** or current working directory.
* You can modify sys.path at runtime or use environment variables like PYTHONPATH.

Example:

import sys

sys.path.append('/path/to/your/package')

import mypackage.module1

**Summary Table**

| **Concept** | **Description** | **Example** |
| --- | --- | --- |
| Module | Single Python .py file | mymath.py |
| Package | Directory with \_\_init\_\_.py and modules | mypackage/ |
| Import module | import module | import mymath |
| Import specific | from module import function | from mymath import add |
| Package import | import package.module or from package import module | import mypackage.module1 |
| \_\_init\_\_.py | Marks directory as a package, can initialize | Importing modules inside package |

If you'd like, I can provide **example code snippets** or help with **creating a sample project** that uses modules and packages!