Dictionaries and Sets Review

Dictionaries

- **Dictionaries**: Dictionaries are built-in data structures that store collections of key-value pairs. Keys need to be immutable data types. This is the general syntax of a Python dictionary:

▼ **Example Code**

```
dictionary = {
    key1: value1,
    key2: value2
}
```

- `dict()` **Constructor**: The `dict()` constructor is an alternative way to build the dictionary. You pass a list of tuples as an argument to the `dict()` constructor. These tuples contain the key as the first element and the value as the second element.

▼ **Example Code**

```
pizza = dict([('name', 'Margherita Pizza'), ('price', 8.9), ('calories_per_slice', 250), ('topp
```

- **Bracket Notation**: To access the value of a key-value pair, you can use the syntax known as bracket notation.

▼ **Example Code**

```
dictionary[key]
```

Common Dictionary Methods

- `get()` **Method**: The `get()` method retrieves the value associated with a key. It's similar to the bracket notation, but it lets you se a default value, preventing errors if the key doesn't exist.

▼ **Example Code**

```
dictionary.get(key, default)
```

- `keys()` **and** `values()` **Methods**: The `keys()` and `values()` methods return a view object with all the keys and values in the dictionary, respectively. A view object is a way to see the content of a dictionary without creating a separate copy of the data.

▼ **Example Code**

```
pizza = {
    'name': 'Margherita Pizza',
    'price': 8.9,
    'calories_per_slice': 250
}

pizza.keys()
# dict_keys(['name', 'price', 'calories_per_slice'])

pizza.values()
# dict_values(['Margherita Pizza', 8.9, 250])
```

- `items()` **Method**: The `items()` method returns a view object with all the key-value pairs in the dictionary, including both the keys and the values.

▼ **Example Code**

- **`clear()` Method**: The `clear()` method removes all the key-value pairs from the dictionary.

▼ Example Code

```
pizza.clear()
```

- **`pop()` Method**: The `pop()` method removes the key-value pair with the key specified as the first argument and returns its value. If the key doesn't exist, it returns the default value specified as the second argument. If the key doesn't exist and the default value i not specified, a `KeyError` is raised.

▼ Example Code

```
pizza.pop('price', 10)
pizza.pop('total_price') # KeyError
```

- **`popitem()` Method**: In Python 3.7 and above, the `popitem()` method removes the last inserted item.

▼ Example Code

```
pizza.popitem()
```

- **`update()` Method**: The `update()` method updates the key-value pairs with the key-value pairs of another dictionary. If they have keys in common, their values are overwritten. New keys will be added to the dictionary as new key-value pairs.

▼ Example Code

```
pizza.update({ 'price': 15, 'total_time': 25 })
```

Looping Over a Dictionary

- **Iterating Over Values**: If you need to iterate over the values in a dictionary, you can write a `for` loop with `values()` to get all the values of a dictionary.

▼ Example Code

```
products = {
    'Laptop': 990,
    'Smartphone': 600,
    'Tablet': 250,
    'Headphones': 70,
}

for price in products.values():
    print(price)
```

Output:

▼ Example Code

```
990
600
250
70
```

- **Iterating Over Keys**: If you need to iterate over the keys in the `products` dictionary above, you can write `products.keys()` or `products` directly.

▼ Example Code

```
# Or

for product in products:
    print(product)
```

Output:

▼ **Example Code**

```
Laptop
Smartphone
Tablet
Headphones
```

- **Iterating Over Key-Value Pairs**: If you need to iterate over the keys and their corresponding values simultaneously, you can iterate over `products.items()`. You get individual tuples with the keys and their corresponding values.

▼ **Example Code**

```
for product in products.items():
    print(product)
```

Output:

▼ **Example Code**

```
('Laptop', 990)
('Smartphone', 600)
('Tablet', 250)
('Headphones', 70)
```

To store the key and value in separate loop variables, you need to separate them with a comma. The first variable stores the key, and the second stores the value.

▼ **Example Code**

```
for product, price in products.items():
    print(product, price)
```

Output:

▼ **Example Code**

```
Laptop 990
Smartphone 600
Tablet 250
Headphones 70
```

- **`enumerate()` Function**: If you need to iterate over a dictionary while keeping track of a counter, you can call the `enumerate()` function. The function returns an `enumerate` object, which assigns an integer to each item, like a counter. You can start the counter from any number, but by default, it starts from 0.

Assigning the index and item to separate loop variables is the common way to use `enumerate()`. For example, with `products.items()`, you can get the entire key-value pair in addition to the index:

▼ **Example Code**

Output:

▼ Example Code

```
0 ('Laptop', 990)
1 ('Smartphone', 600)
2 ('Tablet', 250)
3 ('Headphones', 70)
```

To customize the initial value of the count, you can pass a second argument to `enumerate()`. For example, here we are starting the count from 1.

▼ Example Code

```
for index, product in enumerate(products.items(), 1):
    print(index, product)
```

Output:

▼ Example Code

```
1 ('Laptop', 990)
2 ('Smartphone', 600)
3 ('Tablet', 250)
4 ('Headphones', 70)
```

Sets

- **Sets**: Sets are built-in data structures in Python that do not allow duplicate values. Sets are mutable and unordered, which means that their elements are not stored in any specific order, so you cannot use indices or keys to access them. Also, sets can only contai values of immutable data types, like numbers, strings, and tuples.
- **Defining a Set**: To define a set, you need to write its elements within curly brackets and separate them with commas.

▼ Example Code

```
my_set = {1, 2, 3, 4, 5}
```

- **Defining an Empty Set**: If you need to define an empty set, you must use the `set()` function. Only writing empty curly braces will automatically create a dictionary.

▼ Example Code

```
set() # Set
{}    # Dictionary
```

Common Set Methods

- **`add()` Method**: You can add an element to a set with the `add()` method, passing the new element as an argument.

▼ Example Code

```
my_set.add(6)
```

- **`remove()` and `discard()` Methods**: To remove an element from a set, you can either use the `remove()` method or the `discard()` method, passing the element you want to remove as an argument. The `remove()` method will raise a `KeyError` the element is not found while the `discard()` method will not.

▼ Example Code

- **`clear()` method**: The `clear()` method removes all the elements from the set.

▼ Example Code

```
my_set.clear()
```

Mathematical Set Operations

- **`issubset()` and `issuperset()` Methods**: The `issubset()` and the `issuperset()` methods check if a set is a subset or superset of another set, respectively.

▼ Example Code

```
my_set = {1, 2, 3, 4, 5}
your_set = {2, 3, 4, 5}

print(your_set.issubset(my_set)) # True
print(my_set.issuperset(your_set)) # True
```

- **`isdisjoint()` Method**: The `isdisjoint()` method checks if two sets are disjoint, if they don't have elements in common.

▼ Example Code

```
my_set = {1, 2, 3}
your_set = {4, 5, 6}

print(my_set.isdisjoint(your_set)) # True
```

- **Union Operator (`|`)**: The union operator `|` returns a new set with all the elements from both sets.

▼ Example Code

```
my_set = {1, 2, 3}
your_set = {4, 5, 6}

my_set | your_set # {1, 2, 3, 4, 5, 6}
```

- **Intersection Operator (`&`)**: The intersection operator `&` returns a new set with only the elements that the sets have in common.

▼ Example Code

```
my_set = {1, 2, 3, 4, 5}
your_set = {2, 3, 4, 6}

my_set & your_set # {2, 3, 4}
```

- **Difference Operator (`-`)**: The difference operator `-` returns a new set with the elements of the first set that are not in the other sets.

▼ Example Code

```
my_set = {1, 2, 3, 4, 5}
your_set = {2, 3, 4, 6}

my_set - your_set # {1, 5}
```

- **Symmetric Difference Operator (`^`)**: The symmetric difference operator `^` returns a new set with the elements that are either in the first or the second set, but not both.

```
your_set = {2, 3, 4, 6}

my_set ^ your_set # {1, 5, 6}
```

- `in` **Operator**: You can check if an element is in a set or not with the `in` operator.

▼ **Example Code**

```
print(5 in my_set) # True
```

Python Standard Library

- **Python Standard Library**: A library gives you pre-written and reusable code, like functions, classes, and data structures, that you can reuse in your projects. Python has an extensive standard library with built-in modules that implement standardized solutions for many problems and tasks. Some examples of popular built-in modules are `math`, `random`, `re` (short for "regular expressions" and `datetime`.

Import Statement

- **Import Statement**: To access the elements defined in built-in modules, you use an import statement. Import statements are generally written at the top of the file. Import statements work the same for functions, classes, constants, variables, and any other elements defined in the module.
- **Basic Import Statement**: You can use the `import` keyword followed by the name of the module:

▼ **Example Code**

```
import module_name
```

Then, if you need to call a method from that module, you would use dot notation, with the name of the module followed by the name of the method.

▼ **Example Code**

```
module_name.method_name()
```

For example, you would write the following in your code to import the `math` module and get the square root of 36:

▼ **Example Code**

```
import math

math.sqrt(36)
```

- **Importing a Module with a Different Name**: If you need to import the module with a different name (also known as an "alias"), you can use `as` followed by the alias at the end of the import statement. This is often used for long module names or to avoid naming conflicts.

▼ **Example Code**

```
import module_name as module_alias
```

For example, to refer to the `math` module as `m` in your code, you can assign an alias like this:

▼ **Example Code**

```
import math as m
```

Then, you can access the elements of the module using the alias:

- **Importing Specific Elements**: If you don't need everything from a module, you can import specific elements using `from`. In this case, the import statement starts with `from`, followed by the module name, then the `import` keyword, and finally the names of the elements you want to import.

▼ **Example Code**

```
from module_name import name1, name2
```

Then, you can use these names without the module prefix in your Python script. For example:

▼ **Example Code**

```
from math import radians, sin, cos

angle_degrees = 40
angle_radians = radians(angle_degrees)

sine_value = sin(angle_radians)
cos_value = cos(angle_radians)

print(sine_value) # 0.6427876096865393
print(cos_value)  # 0.766044443118978
```

This is helpful, but it can result in naming conflicts if you already have functions or variables with the same name. Keep it in mind when choosing which type of import statement you want to use.

If you need to assign aliases to these names, you can do so as well, using the `as` keyword followed by the alias.

▼ **Example Code**

```
from module_name import name1 as alias1, name2 as alias2
```

- **Import Statement with Asterisk (`*`)**: The asterisk tells Python that you want to import everything in that module, but you want tc import it so that you don't need to use the name of the module as a prefix.

▼ **Example Code**

```
from module_name import *
```

For example, if you use this to import the `math` module, you'll be able to call any function defined in that module without specifying the name of the module as a prefix.

▼ **Example Code**

```
from math import *
print(sqrt(36))  # 6.0
```

However, this is generally discouraged because it can lead to namespace collisions and make it harder to know where names come from

`if __name__ == '__main__'`

- **`__name__` Variable**: `__name__` is a special built-in variable in Python. When a Python file is executed directly, Python sets the value of this variable to the string `"__main__"`. But if the Python file is imported as a module into another Python script, the valu of the `__name__` variable is set to the name of that module.

This is why you'll often find this conditional in Python scripts. It contains the code that you only want to run **only** if the Python script is running as the main program.

```
# Code
```

Assignment

- [ ] Review the Dictionaries and Sets topics and concepts.

Please complete the assignment

Submit

Ask for Help