# LİSTS-TUPLES-SETS-DICTIONARIES

## Tuples

* Lists in Python are ordered collections of items where each item is assigned an index, starting from 0.
* Tuples, like lists, are ordered collections of items created with parentheses.
* Tuples, like lists, can contain duplicate elements.
* Tuple unpacking allows for assigning tuple items to variables. The values will be assigned in the order they appear in the tuple.

birthday\_date = (12, "August", 1993)

day, month, year = birthday\_date

print(day) //12

print(month) //"August",

print(year) // 1993

* The \* operator in tuple unpacking is used to gather multiple elements from the tuple into a list. This is useful when dealing with tuples of unknown length.

scores = (98, 96, 91, 88, 64)

winner, \*rest = scores

print(winner) //98

print(rest) //[96, 91, 88, 64]

* Listenin bir değeri başka bir değer ile değiştirilirse o indexteki değer artık yeni değer ile değiştirilir ve index kayması olamaz:

years = [2002, 2008, 1999]

years[1] = 2007

for year in years:

  print(year)

Output:

2002

2007

1999

* İndex kayması olması için insert methodu kullanılmalıdır.

years = [2002, 2008, 1999] years.insert(1, 2007) # 1. indekse 2007 ekle print(years) //[2002, 2007, 2008, 1999]

## SETS

* Sets, unlike lists and tuples, are unordered collections. They are created with curly brackets { }.
* Sets are unordered and don't support indexing or slicing.
* Sets can't have duplicates, which is very helpful when developers need to ensure that each item in a collection is unique. For example, in social media apps, your friends list should not have duplicates.
* Adding duplicate items to a set doesn't cause an error; instead, it's ignored.

friends = {'Anna', 'Mery', 'Mery', 'Jonathan'}

print(friends) //{'Anna', 'Mery', 'Jonathan'}

* Like lists and tuples, sets can have values with different data types.
* Sets are mutable, meaning you can add or remove items from them.
* Use the **add()** and **remove()**functions, each with a value as an argument, to add or remove it from a set.

guests = {'Anna', 'Mery', 'Jonathan'}

#adding 'Robert'

guests.add('Robert')

#removing 'Mery'

guests.remove('Mery')

print(guests) //{'Anna', 'Jonathan', 'Robert'}

* The **append()** function works only with ordered collection types, like lists, and adds an item to the end of the collection. Sets are unordered, that's why you can't use it on them.
* The **clear()** function doesn't accept an argument and removes all the items from a set.
* The **union()** function called returns a new set with all elements from both sets, omitting duplicates.

set1 = {'apple', 'banana'}

set2 = {'banana', 'cherry'}

combined\_set = set1.union(set2)

print(combined\_set) //{'cherry', 'apple', 'banana'}

* The **difference()** function returns a set containing elements that are only in the first set and not in the second.

set1 = {'apple', 'banana', 'cherry'}

set2 = {'banana', 'orange'}

unique = set1.difference(set2)

print(unique) // {'cherry', 'apple'}

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

## DICTIONARIES (mutable)

* While **strings** are the most commonly used data type for keys, other immutable types can also serve as keys. Values can be of any data type.
* Lists can’t be used as keys in dictionaries. (Because: Lists are mutable)
* Dictionaries can have duplicate values, but not duplicate keys. Values with duplicate keys will overwrite existing values.

car = {

  "brand": "Audi",

  "model": "Q5",

  "model": "A5"

}

print(car) // {'brand': 'Audi', 'model': 'A5'}

* To access values in dictionaries, you need to use the **keys**.

car = {

  "brand": "Audi",

  "model": "Q5",

  "year": "2008"

}

print(car["brand"]) // Audi

print(car["model"]) // Q5

print(car["year"]) // 2008

* Another way to access values in a dictionary is through the **get()** function.
* It's called on a dictionary using **dot . notation** and accepts the key as an argument
* You can get all the values and keys of a dictionary using the **values()** and **keys()** functions, respectively.

contact = {

  "name": "John",

  "company": "Google",

}

info\_keys = contact.keys()

info\_values = contact.values()

print(info\_keys) // dict\_keys(['name', 'company'])

print(info\_values) // dict\_values(['John', 'Google'])

* The items() function returns all the **key:value** pairs in a dictionary.

car = {

  "brand": "Audi",

  "model": "Q5"

}

info = car.items()

print(info) // dict\_items([('brand', 'Audi'), ('model', 'Q5')])

* You can use keys not only to access values in a dictionary, but also to **change** them.

user = {

  "Name": "Albert",

  "Age": 29

}

user["Age"] = 30

print(user["Age"]) //30

print(user.items()) // dict\_items([('Name', 'Albert'), ('Age', 30)])

* You can add a new item by providing a new key and assigning a value to it.
* The **update()** function updates the dictionary with the items from the given argument.

user = {

  "Name": "Albert",

  "Age": 29

}

# argument: dictionary {"Age": 30}

user.update({"Age": 30})

print(user["Age"]) //30

print(user.items()) // dict\_items([('Name', 'Albert'), ('Age', 30)])

* The update() function can accept dictionaries with multiple items.If an item is new, it will be added to the original dictionary.

user = {

  "Name": "Albert",

  "Age": 29

}

# "Surname": "Johnson" will be added

user.update({"Age": 30, "Surname": "Johnson"})

print(user.items()) //dict\_items([('Name', 'Albert'), ('Age', 30), ('Surname', 'Johnson')])

* The **pop()**function removes the item with the specified key name. It accepts the key of the item you want to remove as an argument.

car = {

  "Brand": "Ford",

  "Model": "Mustang",

  "Color": "red"

}

#removing the item with the "Color" key

car.pop("Color")

print(car) //{'Brand': 'Ford', 'Model': 'Mustang'}

* You can use the **in** operator to check if a key or a value occurs in a dictionary.

car = {

  "Brand": "Ford",

  "Model": "Mustang",

  "Color": "red"

}

print("Color" in car) // True

print("red" in car.values()) // True

* You can iterate through a dictionary using a **for** loop. If you loop through a dictionary, it will return the **keys**.

car = {

  "Brand": "Ford",

  "Model": "Mustang",

  "Color": "red"

}

for i in car:

  print(i)

Oyutput:

Brand

Model

Color

## List Comprehensions

nums = [x for x in range(1,51)]

nums = [x\*2 for x in range(10)]

print(nums) // [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

* You can use a list as the iterable in a list comprehension.

tags = ["travel", "vacation", "journey"]

hashtags = ["#" + x for x in tags]

print(hashtags) // ['#travel', '#vacation', '#journey']

* You can incorporate a condition into a list comprehension, placed after the iterable.

users = ["Brandon", "Emma", "Brian",

"Sophia", "Bella", "Ethan",

"Ava", "Benjamin", "Mia", "Chloe"]

group = [x for x in users if x[0] == "B"]

print(group) // ['Brandon', 'Brian', 'Bella', 'Benjamin']

## Exception Handling

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

* The **try** block holds code that might cause an exception. If an exception occurs, execution of the try block stops, and the **except** block is executed, allowing the program to continue running.

prices = [250, 300, "240", 400]

try:

  #block that may cause an exception

  total = sum(prices)

  print(total)

except TypeError:

  # to perform if there is an exception

  print("Invalid data type")

print("Happy Shopping")

Output:

Invalid data type

Happy Shopping

metin, yazı tipi, ekran görüntüsü, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, yazı tipi, ekran görüntüsü, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu

* When you specify only one type of exception to be handled, other types of exceptions will not be covered. If these other exceptions occur, the program execution will fail.

colors = ['Red', 'Yellow', 'Green']

try:

  #index error

  print(colors[10])

  #wrong exception

except NameError:

  print("Error")

#will not be executed

print("Happy Shopping")

Output:

IndexError: list index out of range

* You can have multiple **except** blocks to handle each possible exception specifically. As a best practice, it is recommended to output a definitive message for each type of handled exception.

colors = ['Red', 'Yellow', 'Green']

try:

  print(colors[10])

except IndexError:

  print("Out of range")

except NameError:

  print("Variable is not defined")

print("Happy shopping")

Output:

Out of range

Happy shopping

* You can choose not to specify the exception type, which allows handling of any exceptions that may occur.

colors = ["Red", "Yellow", "Green"]

try:

  print(colors[10])

except:

  print("Error")

Output:

Error

* You can use the **finally** statement to perform an operation after the try/except block, no matter if an exception occurred or not.

metin, yazı tipi, ekran görüntüsü, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu

prices = [559, 879, "N/A", 349]

try:

  print(sum(prices))

except TypeError:

  print("Check the prices")

finally:

  print("Need help? Contact us")

Output:

Check the prices

Need help? Contact us

* The **else** statement can be used in conjunction with the try/except block and will execute only when no error occurs in the try block.

metin, yazı tipi, ekran görüntüsü, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

books = ['Harry Potter', 'Dune', 'Emma']

try:

  choice = books[1]

except IndexError:

  print("Out of range")

else:

  print(choice + " is a great choice!")

Output:

Dune is a great choice!

* You can trigger your own exceptions based on specific conditions using the **raise** statement. This will immediately stop the program's execution and indicate an error has occurred.

print("Rate from 0 to 10")

rate = 15

if rate > 10 or rate < 0:

  raise ValueError

print("last line")

Output:

Rate from 0 to 10

Traceback (most recent call last):

File "./Playground/file0.py", line 4, in <module>

raise ValueError

ValueError

* To make the exceptions more helpful for the program users you can add a message describing the error.

rating = 15

if rating > 10 or rating < 0:

  raise ValueError("Rate from 0 to 10")

Output:

Traceback (most recent call last):

File "./Playground/file0.py", line 3, in <module>

raise ValueError("Rate from 0 to 10")

ValueError: Rate from 0 to 10