Python is a powerful, high-level, interpreted programming language known for its clear syntax and versatility. It's widely used in web development, data analysis, artificial intelligence, scientific computing, and more.

Data types in Python, demonstrated with code examples:

1. **Numeric Types:** int (integers), float (floating-point numbers), complex (complex numbers).
2. **Boolean Type:** bool (True or False).
3. **Sequence Types:** str (strings), list (mutable, ordered collections), tuple (immutable, ordered collections).
4. **Set Type:** set (mutable, unordered collections of unique items), frozenset (immutable version of a set).
5. **Mapping Type:** dict (unordered collections of key-value pairs).

```
# Python Data Types Demonstration (Separate Print Statements)

# Numeric Types: int, float, complex
int_val = 10
float_val = 3.14
complex_val = 1 + 2j
print(f"Int: {int_val} ({type(int_val)})")
print(f"Float: {float_val} ({type(float_val)})")
print(f"Complex: {complex_val} ({type(complex_val)})")

# Boolean Type: bool
bool_true = True
bool_false = False
print(f"Bool True: {bool_true} ({type(bool_true)})")
print(f"Bool False: {bool_false} ({type(bool_false)})")

# Sequence Types: str, list, tuple
str_val = "Hello"
list_val = [1, 'a']
tuple_val = (1, 'b')
print(f"String: '{str_val}' ({type(str_val)})")
print(f"List: {list_val} ({type(list_val)})")
print(f"Tuple: {tuple_val} ({type(tuple_val)})")

# Set Type: set
set_val = {1, 2, 3}
print(f"Set: {set_val} ({type(set_val)})")

# Mapping Type: dict
dict_val = {"key": "value"}
print(f"Dictionary: {dict_val} ({type(dict_val)})")
```

```
Int: 10 (<class 'int'>)
Float: 3.14 (<class 'float'>)
Complex: (1+2j) (<class 'complex'>)
Bool True: True (<class 'bool'>)
Bool False: False (<class 'bool'>)
String: 'Hello' (<class 'str'>)
List: [1, 'a'] (<class 'list'>)
Tuple: (1, 'b') (<class 'tuple'>)
Set: {1, 2, 3} (<class 'set'>)
Dictionary: {'key': 'value'} (<class 'dict'>)
```

List

A list is a fundamental data structure that allows you to store an ordered collection of items. What makes lists so powerful is that they are mutable, meaning you can change, add, or remove elements after the list has been created. They can also hold items of different data types.

```
# Python List: A Simple Example

# 1. Creating a list
my_simple_list = ["apple", "banana", "cherry", 123, True]
print(f"Original list: {my_simple_list}")
print(f"Type: {type(my_simple_list)}")
print(f"Length: {len(my_simple_list)}")

# 2. Accessing elements (lists are zero-indexed)
print(f"First element: {my_simple_list[0]}") # 'apple'
print(f"Last element: {my_simple_list[-1]}") # True

# 3. Changing an element (mutable)
my_simple_list[1] = "grape"
print(f"List after changing an element: {my_simple_list}")

# 4. Adding elements
my_simple_list.append("orange") # Adds to the end
print(f"List after appending: {my_simple_list}")

my_simple_list.insert(1, "kiwi") # Inserts at a specific position
print(f"List after inserting: {my_simple_list}")

# 5. Removing elements
my_simple_list.remove("cherry") # Removes the first occurrence of a value
print(f"List after removing 'cherry': {my_simple_list}")

# Pop an element (removes and returns the last item by default)
popped_fruit = my_simple_list.pop()
print(f"List after popping last element: {my_simple_list}, Popped: {popped_fruit}")

# 6. Iterating through a list
print("\nElements in the list:")
for item in my_simple_list:
    print(item)
```

```
Original list: ['apple', 'banana', 'cherry', 123, True]
Type: <class 'list'>
Length: 5
First element: apple
Last element: True
List after changing an element: ['apple', 'grape', 'cherry', 123, True]
List after appending: ['apple', 'grape', 'cherry', 123, True, 'orange']
List after inserting: ['apple', 'kiwi', 'grape', 'cherry', 123, True, 'orange']
List after removing 'cherry': ['apple', 'kiwi', 'grape', 123, True, 'orange']
List after popping last element: ['apple', 'kiwi', 'grape', 123, True], Popped: orange

Elements in the list:
apple
kiwi
grape
123
True
```

Tuple

tuple is an ordered collection of items, similar to a list. However, the key difference is that tuples are immutable, meaning once a tuple is created, its elements cannot be changed, added, or removed. Tuples are defined by enclosing elements in parentheses (), and they can also hold items of different data types.

```
# Python Tuple: A Simple Example

# 1. Creating a tuple
my_tuple = ("apple", "banana", "cherry", 123, False)
print(f"Original tuple: {my_tuple}")
print(f"Type: {type(my_tuple)}")
print(f"Length: {len(my_tuple)}")

# 2. Accessing elements (tuples are zero-indexed)
print(f"First element: {my_tuple[0]}") # 'apple'
print(f"Last element: {my_tuple[-1]}") # False

# 3. Attempting to change an element (will cause an error - tuples are immutable)
# Uncomment the line below to see the TypeError
# my_tuple[1] = "grape"
# print(f"Tuple after attempting change: {my_tuple}")

# 4. Concatenating tuples
another_tuple = (4, 5, 6)
combined_tuple = my_tuple + another_tuple
print(f"Combined tuple: {combined_tuple}")

# 5. Counting elements
count_banana = my_tuple.count("banana")
print(f"Number of 'banana' in tuple: {count_banana}")

# 6. Finding an element's index
index_cherry = my_tuple.index("cherry")
print(f"Index of 'cherry': {index_cherry}")

# 7. Iterating through a tuple
print("\nElements in the tuple:")
for item in my_tuple:
    print(item)
```

```
Original tuple: ('apple', 'banana', 'cherry', 123, False)
Type: <class 'tuple'>
Length: 5
First element: apple
Last element: False
Combined tuple: ('apple', 'banana', 'cherry', 123, False, 4, 5, 6)
Number of 'banana' in tuple: 1
Index of 'cherry': 2

Elements in the tuple:
apple
banana
cherry
123
False
```

Dictionary

a dictionary is an unordered collection of data values used to store data in a key:value pair format. Unlike sequences (like lists or tuples), which are indexed by a range of numbers, dictionaries are indexed by keys. These keys must be unique and immutable (like strings, numbers, or tuples), while the values can be of any data type and can be changed. Dictionaries are mutable, meaning you can add, remove, or modify key-value pairs after creation.

```
# Python Dictionary: A Simple Example

# 1. Creating a dictionary
# Keys are unique strings, values are various data types
my_dict = {"name": "Alice", "age": 30, "city": "New York", "is_student": False}
print(f"Original dictionary: {my_dict}")
print(f"Type: {type(my_dict)}")
print(f"Number of key-value pairs: {len(my_dict)}")

# 2. Accessing values by key
print(f"Name: {my_dict["name"]}")
print(f"Age: {my_dict["age"]}")

# Using .get() for safer access (returns None if key not found, or a default value)
print(f"Country (using .get()): {my_dict.get("country", "Not specified")}")

# 3. Modifying a value
my_dict["age"] = 31
print(f"Dictionary after changing age: {my_dict}")

# 4. Adding a new key-value pair
my_dict["occupation"] = "Engineer"
print(f"Dictionary after adding occupation: {my_dict}")

# 5. Removing a key-value pair
# Using del statement
del my_dict["is_student"]
print(f"Dictionary after deleting 'is_student': {my_dict}")

# Using .pop() method (removes key and returns its value)
removed_city = my_dict.pop("city")
print(f"Dictionary after popping 'city': {my_dict}, Removed city: {removed_city}")
```

```
Original dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York', 'is_student': False}
Type: <class 'dict'>
Number of key-value pairs: 4
Name: Alice
Age: 30
Country (using .get()): Not specified
Dictionary after changing age: {'name': 'Alice', 'age': 31, 'city': 'New York', 'is_student': Fals
Dictionary after adding occupation: {'name': 'Alice', 'age': 31, 'city': 'New York', 'is_student':
Dictionary after deleting 'is_student': {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupati
Dictionary after popping 'city': {'name': 'Alice', 'age': 31, 'occupation': 'Engineer'}, Removed c
```

Double-click (or enter) to edit

Set

A Set is an unordered collection of unique items. This means that each element within a set must be distinct; duplicates are automatically removed. Sets are mutable, allowing you to add or remove items. They are defined by enclosing elements in curly braces {}.

```
# Python Set: A Simple Example

# 1. Creating a set (duplicates are automatically removed)
my_set = {1, 2, 3, 2, 1, 'apple', 'banana'}
print(f"Original set: {my_set}")
print(f"Type: {type(my_set)}")
print(f"Length: {len(my_set)}")

# 2. Adding elements to a set
my_set.add('cherry')
print(f"Set after adding 'cherry': {my_set}")

# Attempting to add an existing element (no change occurs)
my_set.add('apple')
print(f"Set after attempting to add 'apple' again: {my_set}")

# 3. Removing elements from a set
my_set.remove(3) # Removes the element 3
print(f"Set after removing 3: {my_set}")

# 4. Checking for presence
print(f"Is 'banana' in the set? {'banana' in my_set}")
print(f"Is 'grape' in the set? {'grape' in my_set}")

# 5. Iterating through a set
print("\nElements in the set:")
for item in my_set:
    print(item)
```

```
Original set: {'apple', 1, 2, 3, 'banana'}
Type: <class 'set'>
Length: 5
Set after adding 'cherry': {'apple', 1, 2, 3, 'cherry', 'banana'}
Set after attempting to add 'apple' again: {'apple', 1, 2, 3, 'cherry', 'banana'}
Set after removing 3: {'apple', 1, 2, 'cherry', 'banana'}
Is 'banana' in the set? True
Is 'grape' in the set? False

Elements in the set:
apple
1
2
cherry
banana
```

Object-Oriented Programming

Object-Oriented Programming (OOP) and its main concepts simply. OOP is a way of organizing code that revolves around objects. Think of an object as a self-contained unit that has both data (what it 'knows') and behavior (what it 'can do'). It helps make code more modular, reusable, and easier to manage, especially in large projects.

The core concepts of OOP, often called its four pillars, are:

1. Encapsulation: This is about bundling data (attributes) and the methods (functions) that operate on that data into a single unit, which is typically a class. It's like putting all related things into a capsule. The idea is to hide the internal workings of an object and only expose what's necessary to interact with it.
2. Inheritance: This allows a new class (called a 'child' or 'subclass') to derive or inherit properties (attributes) and behaviors (methods) from an existing class (called a 'parent' or 'superclass'). It promotes code reusability because you don't have to write the same code again.
3. Polymorphism: This literally means 'many forms'. It's the ability of different objects to respond to the same message (method call) in their own unique way. This means you can treat objects of different types uniformly.
4. Abstraction: This involves simplifying complex systems by focusing on the essential features and hiding unnecessary details. It concentrates on 'what' an object does rather than 'how' it does it.

```
# Simplest OOP Example: A 'Dog' Class

# Define a Class (a blueprint for creating objects)
class Dog:
    # The constructor method: initializes new 'Dog' objects
    def __init__(self, name, age):
        self.name = name  # Attribute: the dog's name
        self.age = age    # Attribute: the dog's age

    # A method: a function that belongs to the Dog class
    def bark(self):
        return f"{self.name} says Woof!"

# Create Objects (instances) from the Dog class
my_dog = Dog("Buddy", 3)
friend_dog = Dog("Lucy", 5)

# Access attributes and call methods on the objects
print(f"My dog's name is {my_dog.name} and he is {my_dog.age} years old.")
print(my_dog.bark())

print(f"Friend's dog's name is {friend_dog.name} and she is {friend_dog.age} years old.")
print(friend_dog.bark())
```

```
My dog's name is Buddy and he is 3 years old.
Buddy says Woof!
Friend's dog's name is Lucy and she is 5 years old.
Lucy says Woof!
```

Start coding or generate with AI.