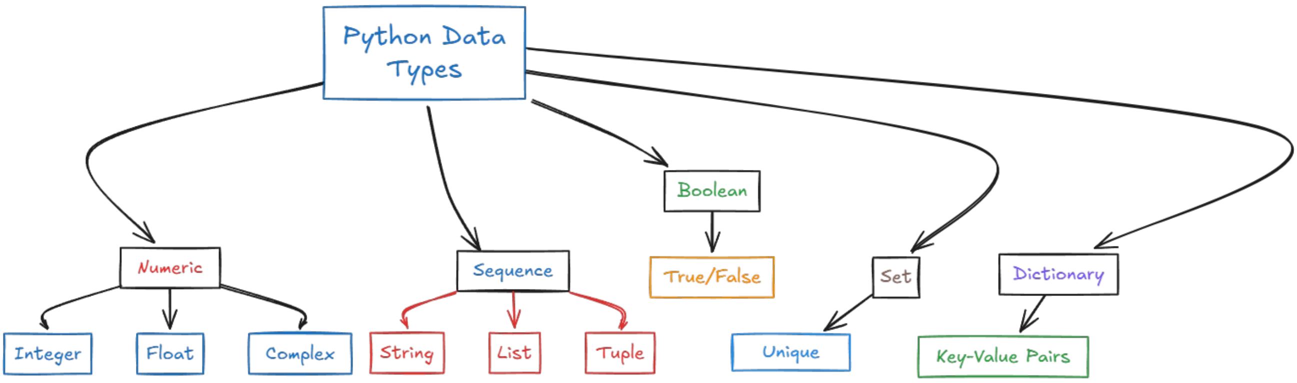
Data Structures

A **data structure** is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently.

In Python, data structures are built-in or custom-coded arrangements—like lists, dictionaries, tuples, sets, stacks, queues, and deques—that help manage collections of data for different types of tasks.

Data structures are fundamental to programming in Python and everywhere else.



Number

Numbers are Python’s data type for math and counting. There are two main kinds:

Integers (int): Whole numbers, positive or negative, like 7, -3, or 42.

Floating-point numbers (float): Numbers with decimals, like 5.0, -2.75, or 3.14159.

> **Real-Life Example:** counting apples (integers): apples = 5 & measuring a distance (floats): distance_km = 23.7

Use Cases: Scores, prices, measurements, quantity, temperatures.

In [1]:

```
distance_km = 23.7
apples = 5
print(apples,"&",distance_km)
```

```
5 & 23.7
```

List

An ordered, mutable(changeable) collection. Stores items of any type and allows duplicates.

> **Real-Life Example:** Shopping list: You have a list for groceries—order matters, you can add or remove items, and even jot down "milk" twice if you want.

> Can add/remove/reorder items.

> Items can repeat (duplicate values allowed).

In [2]:

```
groceries = ["milk", "eggs", "cheese", "bread"]
groceries
```

Out[2]:

```
['milk', 'eggs', 'cheese', 'bread']
```

Tuple

An ordered, immutable(unchangeable) collection. Once created, it can't be changed.

> **Real-Life Example:** Your birth date: (day, month, year)—this won't change over time, just like a tuple can’t be modified after it's created.

> Fixed size once made.

> Useful for safe storage of related, unchangeable data

In [3]:

```
birth_date = (15, "August", 1999)
birth_date
```

Out[3]:

```
(15, 'August', 1999)
```

Set

An unordered collection of unique items. Mutable, but no duplicates allowed.

> **Real-Life Example:** Club membership roster: Only unique names—no one can join twice, and order is irrelevant.

> Order doesn’t matter.

> Good for things like class attendance.

In [4]:

```
members = {"Alice", "Bob", "Charlie"}
members
```

Out[4]:

```
{'Alice', 'Bob', 'Charlie'}
```

Dictionary

An unordered, mutable collection of key-value pairs.

> **Real-Life Example:** Contacts app: You search by name (key) to find a phone number (value).

> Keys are unique.

> You can quickly look up data by its key, not by position.

In [5]:

```
contacts = {"Alice": "555-1234", "Bob": "555-5678"}
contacts
```

Out[5]:

```
{'Alice': '555-1234', 'Bob': '555-5678'}
```

String

An immutable(unchangeable) sequence of characters.

> **Real-Life Example:** A written sentence: "Hello, world!"—you read left to right, character by character.

> Each character can be accessed by position.

> Strings can't be changed after creation.

In [6]:

```
greeting = "Hello, world!"
greeting
```

Out[6]:

```
'Hello, world!'
```

Boolean

A boolean is a simple data type in Python that can have one of two values: >**True**, **False**

Think of booleans as answering yes/no questions in your code—either something is true, or it is not.

In [7]:

```
# Checking Adulthood
# Scenario: Want to see if someone is old enough to vote.
age = 20
is_eligible = age >= 18
print(is_eligible)  # Output: True
# Analogy: Is this person old enough? Yes (True) or No (False).
```

```
True
```

---

frozenset

A frozen set (frozenset) in Python is just like a regular set—it holds unique, unordered items—but it's immutable: you can't add, remove, or change anything after it's created.

> Unordered, unique items (like a set)

> Cannot be changed after creation (unlike a regular set)

> Can be used as a dictionary key (unlike a regular set)

> Useful for fixed collections—like valid school grades, RGB colors, or a list of founding members that shouldn't change

In [8]:

```
team = frozenset(["Alice", "Bob", "Charlie"])
print(team)  # frozenset({'Alice', 'Bob', 'Charlie'})
# team.add("Dana")  # Error: cannot add to a frozenset
```

```
frozenset({'Alice', 'Charlie', 'Bob'})
```

Stack

A stack follows the LIFO (Last-In, First-Out) rule: the last item added is the first to be removed. Imagine a stack of plates—each new plate goes on top, and you always remove the top one first.

> Each append() records a new action.

> Each pop() undoes the most recent action.

> **Real-Life Example**: **Undo** in Text Editors Whenever you use Undo (Ctrl+Z) in tools like Word or Google Docs, every user action is stored. The last change you made is the first to be "undone," just as in a stack.

In [9]:

```
# Stack to store actions
actions = []

# Perform some actions
actions.append('Type A')
actions.append('Type B')
actions.append('Delete B')

# Undo operations
print("Undo:", actions.pop())  # 'Delete B'
print("Undo:", actions.pop())  # 'Type B'
```

```
Undo: Delete B
Undo: Type B
```

Queues

A queue operates on the FIFO (First-In, First-Out) principle: the first item added is the first to be taken out. Think of people waiting in a line; whoever arrives first gets served first.

> append() adds a caller to the end.

> popleft() removes the caller who's waited longest.

> **Real-Life Example**: **Customer Service Call Center** When customers call a support number, their calls are handled in the order received—no matter how many join the queue.

In [10]:

```python
from collections import deque

# Create a queue to represent print jobs
printer_queue = deque()

# Simulate users sending print jobs
printer_queue.append("Report.pdf")
printer_queue.append("Invoice.docx")
printer_queue.append("Presentation.pptx")

print("Printing:", printer_queue.popleft())  # Report.pdf
print("Printing:", printer_queue.popleft())  # Invoice.docx
print("Printing:", printer_queue.popleft())  # Presentation.pptx
```

```
Printing: Report.pdf
Printing: Invoice.docx
Printing: Presentation.pptx
```

Deque

A deque allows you to append and pop from the front and back with equal efficiency. It can behave as a stack (last-in, first-out, or LIFO), a queue (first-in, first-out, or FIFO), or anything in between, depending on which methods you use.

> append(item): Add an item to the right (rear) end.

> appendleft(item): Add an item to the left (front) end.

> pop(): Remove and return the rightmost item.

> popleft(): Remove and return the leftmost item

Real Life Example: Browser History Navigation When you browse the web, your back and forward history is managed like a deque:

Back button: As you visit sites, each gets pushed to the right end.

Forward button: When you move back, you might add sites to the left.

Deque allows near-instant updates on both ends, perfect for navigating backwards and forwards efficiently.

In [11]:

```python
from collections import deque

d = deque()

# Append (enqueue/push)
d.append('A')          # Add to end
d.appendleft('Start') # Add to front

# Pop (dequeue/pop)
print(d.pop())         # Remove from end (stack pop)
print(d.popleft())     # Remove from front (queue dequeue)
```

```
A
Start
```

Type conversion

Type casting (type conversion) is the process of converting data from one type to another in Python. There are two main types:

> **Implicit type conversion:** Python handles conversions automatically when safe and possible.

> **Explicit type conversion (type casting)**: The programmer manually changes a value to a new type using constructor functions.

> Python supports type casting among many of its built-in types using functions such as int(), float(), str(), list(), tuple(), set(), dict(), etc.

> Not every data type can be converted to every other type—compatibility and format rules apply.

Input Returns String

In [12]:
```
num = input("Enter a Num :")
type(num)
```

Out[12]:
```
str
```

Boolean --> String

In [13]:
```
bool_ = True
str_bool = str(bool_)
print("Success:"+ str_bool)
type(str_bool)
```

```
Success:True
```

Out[13]:
```
str
```

Boolean --> int

In [14]:
```
bool_ = True
str_bool = int(bool_)
print("Success:", str_bool)
type(str_bool)
```

```
Success: 1
```

Out[14]:
```
int
```

Int --> Boolean

In [15]:
```
integer = 14 # Any non-zero value returns true
int_bool = bool(bool_)
print("Success:",int_bool)
type(int_bool)
```

```
Success: True
```

Out[15]:
```
bool
```

String(Whole Number) --> Int

In [16]:
```
num1 = int(num)
type(num1)
```

Out[16]:
```
int
```

In [17]:
```
print(num1)
```

```
19
```

String(String) --> Int --> Error

In [18]:
```
str1 = "Hello"
type(str1)
```

Out[18]:
```
str
```

In [19]:
```
str_int = int(str1)
type(str_int)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[19], line 1
----> 1 str_int = int(str1)
      2 type(str_int)

ValueError: invalid literal for int() with base 10: 'Hello'
```

String(String) --> Float --> Error

```
In [20]: str2 = "Hello"
         type(str2)

Out[20]: str
```

```
In [21]: str_f = float(str2)
         type(str_f)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 1
----> 1 str_f = float(str2)
      2 type(str_f)

ValueError: could not convert string to float: 'Hello'
```

String(float) --> Int --> Error

```
In [23]: numf = input("Enter a Num :")
         type(numf)

Out[23]: str
```

```
In [24]: numff = int(numf)
         type(numff)

Out[24]: int
```

String(float) --> float --> Int

```
In [25]: numff = int((float(numf)))
         type(numff)

Out[25]: int
```

Adding String & Int --> Error

```
In [26]: num_int = 7
         type(num_int)

Out[26]: int
```

```
In [27]: num_str= "5"
         type(num_str)

Out[27]: str
```

```
In [28]: add = num_int + num_str
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 add = num_int + num_str

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Solution --> Convert(String)--> Int

```
In [29]: num_str_conv = int(num_str)
         type(num_str_conv)

Out[29]: int
```

```
In [30]: add = num_int + num_str_conv
```

```
In [31]: add

Out[31]: 12
```

Adding String & String --> Concat

In [33]:

```
str1= input("Input a String")
str2= input("Input a String")
concat = str1 + str2
print(concat)
```

```
SUMIT SINGH
```

Int --> float

In [34]:

```
Num = 100
f = float(Num)
f
```

Out[34]:

```
100.0
```

Int --> String

In [35]:

```
print("I have got" + Num + "Numbers in Exam")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 print(             + Num + "Numbers in Exam")

TypeError: can only concatenate str (not "int") to str
```

In [36]:

```
string = str(Num)
print("I have got " + string + " Numbers in Exam")
```

```
I have got 100 Numbers in Exam
```

Int --> Hexadecimal

In [37]:

```
Num = 100
hexa = hex(Num)
hexa
```

Out[37]:

```
'0x64'
```

Int --> Complex

In [38]:

```
complexx = complex(Num)
complexx
```

Out[38]:

```
(100+0j)
```

Int --> Octal

In [39]:

```
octa = oct(Num)
octa
```

Out[39]:

```
'0o144'
```

Int --> Binary

In [40]:

```
Binary = bin(Num)
Binary
```

Out[40]:

```
'0b1100100'
```

String --> Unicode -->ERROR

In [41]:

```
char = "abc"
uni = ord(char)
uni
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[41], line 2
      1 char = "abc"
----> 2 uni = ord(char)
      3 uni

TypeError: ord() expected a character, but string of length 3 found
```

String(length of 1) --> Unicode -->CORRECT

In [42]:

```
char = "a"
uni = ord(char)
uni
```

Out[42]:

```
97
```

Unicode --> char

In [43]:

```
numchar = chr(Num)
char
```

Out[43]:

```
'a'
```

THANKS
