Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #5 - Sequences

Overview

- The need for more than ints and floats
- Sequences
- Lists
- Tuples
- Strings

Problem

- Problem: Tracking Quiz Retakes
- Scenario:
  - A teacher wants to track how many times each student retook a quiz.
  - For example:
    - Alice retook it 2 times
    - Bruno retook it 1 time
    - Carla retook it 3 times
- The teacher wants to:
  - Record the number of retakes for each student.
  - Print a summary showing each student and their retake count.
  - Later, decide who needs extra help (e.g., students with 2+ retakes).

Possible solution (pseudo code)

Algorithm TrackRetakes

Define a group of students

For each student in the group:
    Ask how many times they retook the quiz
    Store this information in a sequence

For each entry in the sequence:
    Display the student's name and their retake count

Identify students with high retake counts
Display their names as needing extra support

End Algorithm

What do we need?

- Define a group of …
- Store information in a sequence
- …
- In summary: a way (or ways) of storing multiple values with a common name

Before we start our topic …

SOME USEFUL INFORMATION

Everything is an object

- Every value in Python is an object
  - from numbers and strings to functions and modules
- Each object has:
  - ID: that identifies it during its lifetime
  - Content: the actual data it holds
  - Type: the class that defines its behavior and operations

```
type(42)          # <class 'int'>
type("hello")     # <class 'str'>
type(None)        # <class 'NoneType'>
```

type()

- The **type()** function returns the **type of an object**.
- It’s useful for checking what kind of data you're working with
  - especially when debugging or validating input.

```
print(type(42))           # int
print(type(3.14))         # float
print(type(2 + 3j))       # complex
print(type("hello"))      # str
print(type(True))         # bool
print(type(None))         # NoneType
```

Types you know

- int
- float
- bool
- None
- string

- All can only store a single value
  - Except strings, if we consider they are composed by characters

id()

- In Python, **every object has a unique ID** that identifies it during its lifetime.

- You can retrieve it using the built-in **id()** function.

```
num = 10
x = 19.1
s = "Fundamentals"

print(f"id(num) = {id(num)}")
print(f"id(x) = {id(x)}")
print(f"id(s) = {id(s)}")
```

```
id(num) = 140725449065672
id(x) = 2768649368496
id(s) = 2768652647280
```

Functions and methods

- Function
  - Is a block of code that performs a specific task.
  - It can be defined anywhere
    - usually at the top level of a program or module

```
def greet(name):
    print("Hello,", name)

# calling the function
greet("Miguel")
```

- Method
  - Is a function that belongs to an object (or class).
  - It is called using dot notation
    - because it operates on data stored in that object.

```
class Person:
  def greet(self):
    print("I'm a person!")

p = Person()
p.greet()
# 'greet' is a method
```

Mutable?

- **Mutable** refers to something that is capable of change.
- If an object or entity is mutable, it means its state or content can be altered after it is created.

- Examples in everyday life:
  - Our mood is mutable
    - it can shift throughout the day.
  - A document is mutable if you can edit it.

- In Python, **mutable objects** are those whose internal state (contents) can be changed after they are created
  - *without changing their identity*.

COLLECTIONS AND SEQUENCES

Collection

- Multiple values, a single name
  - Example:

- In programming:
  - Single “variable” used to store multiple values

Sequences

- In programming languages, **sequences** are **ordered collections of elements**
  - that can be accessed, manipulated, and iterated over.

- They’re fundamental structures used to group related data and perform operations on it efficiently

Why we need sequences?

- They help you:
  - Store multiple values under one variable name
  - Access elements by position
  - Perform batch operations
    - e.g., filtering, mapping, sorting
  - Build scalable and readable code

Key Characteristics of Sequences

- **Ordered**:
  - Elements have a defined position (index)
    - starting from 0

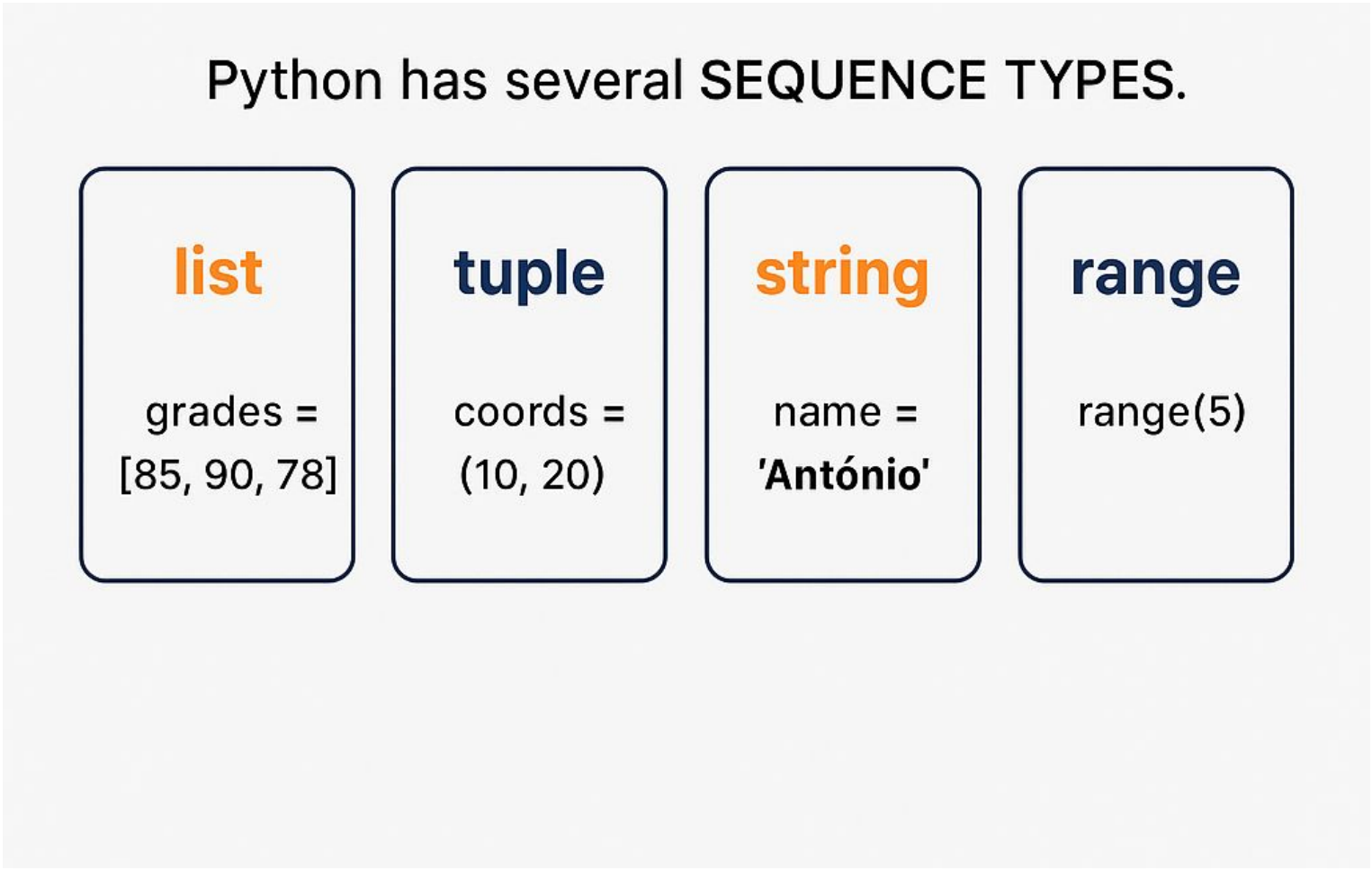
- **Iterable**:
  - You can iterate through them using constructs like for loops.

Key Characteristics of Sequences

- Mutable or Immutable:
  - Some sequences can be changed (like Python lists)
  - others cannot (like Python tuples or strings).

- Homogeneous or Heterogeneous:
  - Some languages enforce one data type per sequence;
  - others (like Python) allow mixed types.

Sequences


Python has several SEQUENCE TYPES.
list
grades = [85, 90, 78]
tuple
coords = (10, 20)
string
name = 'António'
range
range(5)

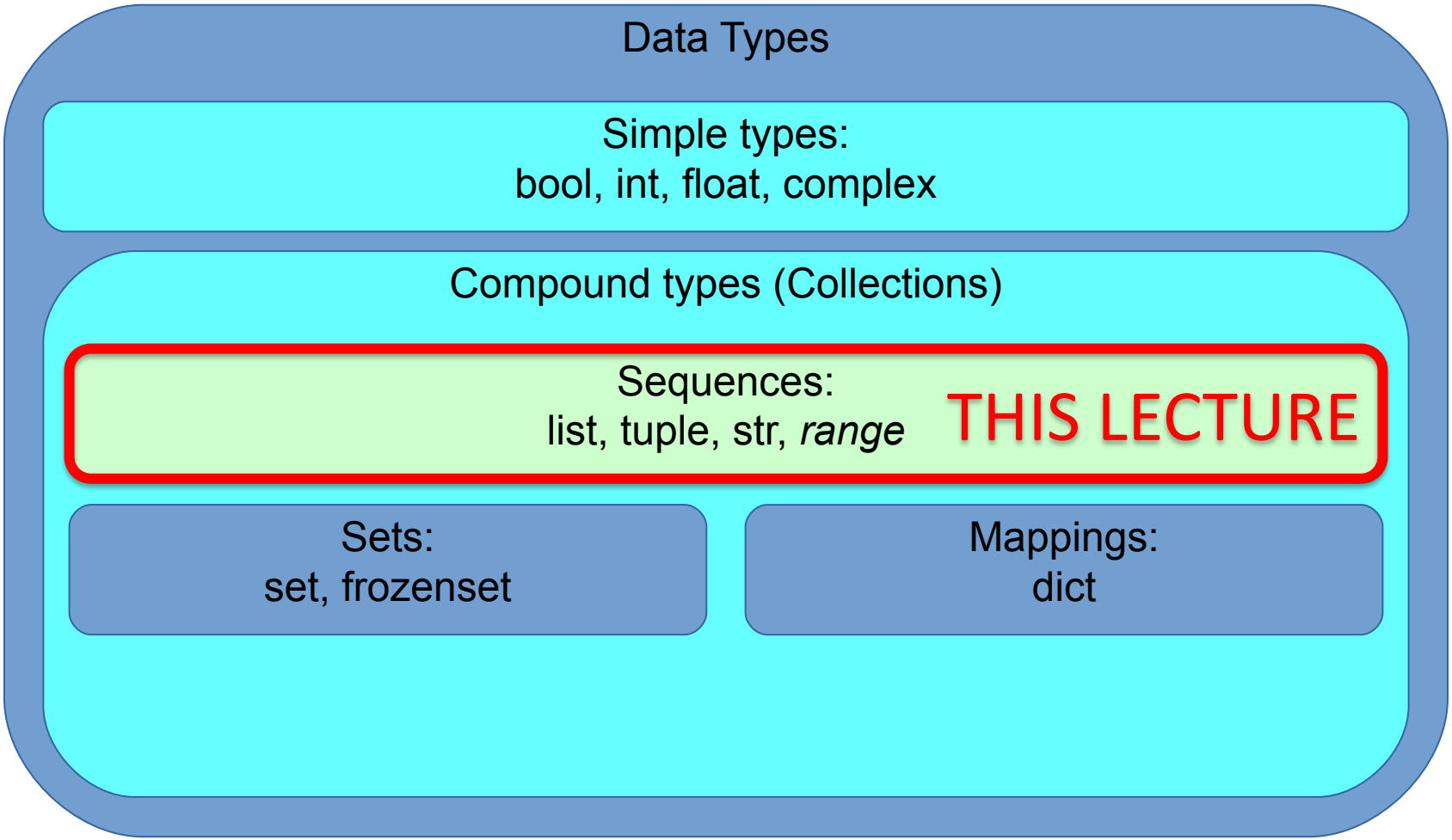
Examples of sequences in python

Examples in Python

Sequence Type	Description	Example
`list`	Mutable, ordered collection	`grades = [85, 90, 7`
`tuple`	Immutable, ordered collection	`coords = (10, 20)`
`str`	Immutable sequence of characters	`name = "António"`
`range`	Sequence of numbers, often used in loops	`range⑤` → `0, 1, 2, 3, 4`

IN PYTHON

Python types



PYTHON LISTS

Lists

- A list is a **mutable sequence of values** of any type.
- The values in the list are called elements or items.

- They are defined using brackets.

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
empty = []                              # an empty list
things = ['spam', 2.0, [1, 2]]  # a list inside a list!
```

Play ▸

What can we do with lists?

- Obtain length, sum, etc using built-in functions
- Accessing the elements
  - One or more
- Change the values
- Add / remove elements
- …

Functions

```
numbers = [10, 20, 30, 40]
empty = []                          # an empty list
things = ['spam', 2.0, [1, 2]]      # a list inside a list!
```

- Function **len()** returns the length of a collection.

```
len(numbers)    #-> 4
len(empty)      #-> 0
len(things)     #-> 3
```

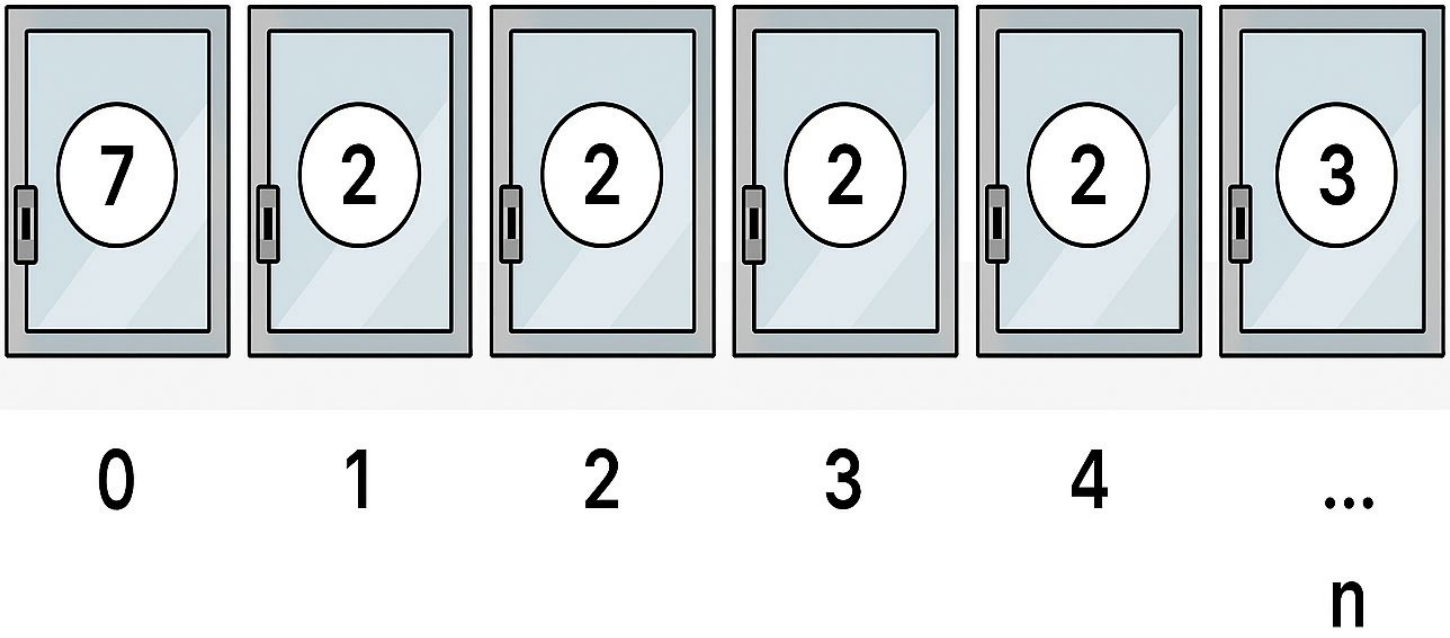
Play ▸

- Other built-in functions apply to sequences.

```
sum(numbers)    #-> 100
min(numbers)    #-> 10
max(numbers)    #-> 40
```

Accessing by position / Indexing

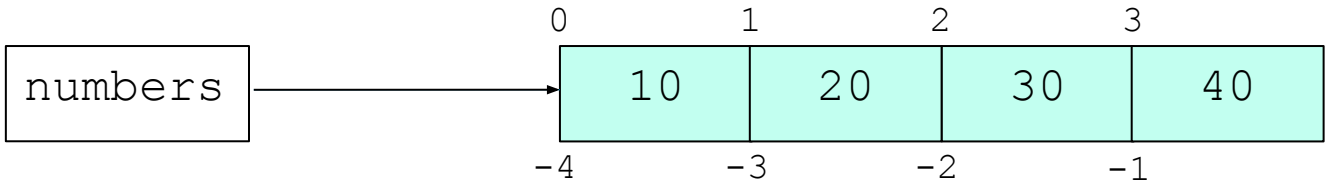
The basic idea:



Accessing elements / Indexing

- We can **access each element** of a sequence using the bracket operator and a value – the index.

```
numbers[0]      #-> 10          (index starts at 0)
fruits[2]       #-> 'orange'
```



- A negative index counts backward from the end.

```
numbers[-1]     #-> 40
```

Play ▸

Accessing elements / indexing

- Any integer expression may be used as an index.

```
numbers[(9+1)%4]   #-> 30
```

- Using an index outside the list bounds is an error.

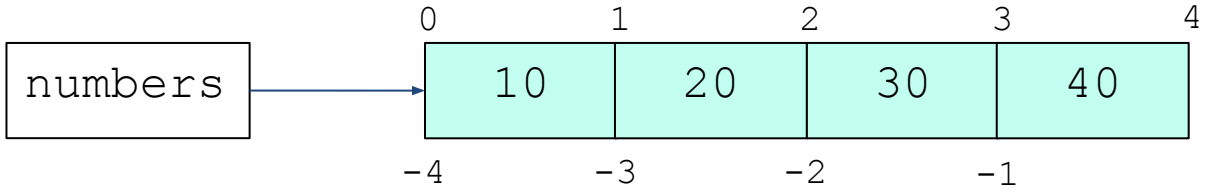
```
numbers[4]     #-> IndexError
numbers[-5]    #-> IndexError
```

Extracting subsequences / Slicing

- We can extract a subsequence using **slicing**.

- Considering one previous example:

```
numbers = [10, 20, 30, 40]
```

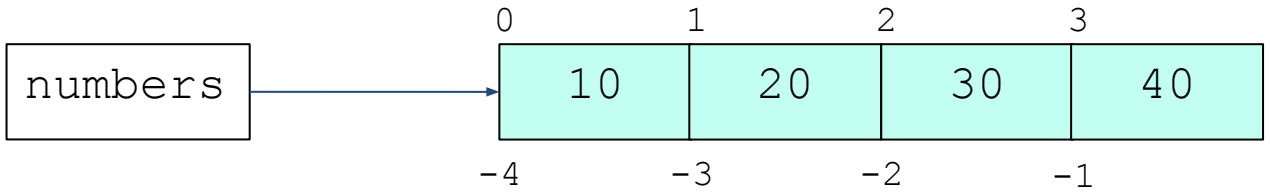



```
numbers[1:3]
#-> [20, 30]

numbers[0:4:2]
#-> [10, 30] (step = 2)

numbers[2:2]
#-> []
```

Extractig subsequences / Slicing



- Negative indices may be used too.

```
numbers[-4:-2]   #-> [10, 20]
numbers[1:-1]    #-> [20, 30]
```

- Indices may be omitted for the start or end.

```
numbers[:2]   #-> [10, 20]
numbers[3:]   #-> [40]
numbers[:]    # a full copy of numbers
```

Play ▸

Changing list content

- As lists are **mutable,** we can change their content.

```
numbers[1] = 99
numbers        #-> [10, 99, 20, 40]
```

- We can even change a sublist.

```
numbers[2:3] = [98, 97]
numbers        #-> [10, 99, 98, 97, 40]
```

Play ▸

Changing list content

- Lists have several methods to change their contents.

```
lst = [1, 2]
lst.append(3) # appends 3 to end of lst → [1, 2, 3]

x = lst.pop()   # removes element with higher index
                # lst → [1, 2], x → 3


lst.extend([4, 5])   # lst → [1, 2, 4, 5]
lst.insert(1, 6)     # lst → [1, 6, 2, 4, 5]
x = lst.pop(0)       # lst → [6, 2, 4, 5], x → 1
```

Play ▸

List methods

Play ▸

`lst.append(item)`	Add item to the end
`lst.insert(pos, item)`	Insert item at the given position
`lst.extend(collection)`	Add all the items in the argument
`lst.pop()`	Remove last item
`lst.pop(pos)`	Remove item in given position
`lst.remove(item)`	Remove first occurence of given item (if any)
`lst.sort()`	Sort the items in the list
`lst.reverse()`	Reverse the order of items in the list
`lst.index(item)`	Position of first occurrence of given item
`lst.count(item)`	Number of occurrences of given item

Mutable sequence operations (Python documentation)

List methods in action

```
lst = ["a", "b", "c"]

# Adding items
lst.append("d")          # add item to the end
lst.insert(2, "x")       # insert item at the given position
lst.extend(["y", "z"])   # add all the items in the argument
                         # equivalent to: lst += ["y", "z"]

# Removing items
r1 = lst.pop()           # remove and return the last item
r2 = lst.pop(1)          # remove and return the item at position

lst2 = list("abracadabra")
del lst2[0]
del lst2[1:3]

lst2.remove("a")         # remove first occurrence of item "a"
```

List methods in action

```
lst2 = list("abracadabra")


# Finding items
idx1 = lst2.index("a")  # position of first occurrence of item

# Counting items
num = lst2.count("a")   # number of occurrences of item

# Sorting
lst2.sort()

# Reversing
lst2.reverse()
```

Play ▸

Displaying list content / Traversing

- We already know how to access to list elements.
- Displaying its value is as simple as using print()

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
print(fruits[1])
```

- To display all the content (elements) of a list **we only need to iterate** through all its positions.
- The most common way to traverse the elements of a sequence is with a for loop.

```
for fruit in fruits:
    print(fruit)
```

banana
pear
orange

Play ▸

Displaying list content / Traversing

- We may also traverse using the indexes.

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

- In this case, we may use a **`while`** loop instead.

```
i = 0
while i < len(fruits):
    print(i, fruits[i])
    i += 1
```

- It is also possible to traverse the indexes and items simultaneously.

```
for i, f in enumerate(fruits):
    print(i, f)
```

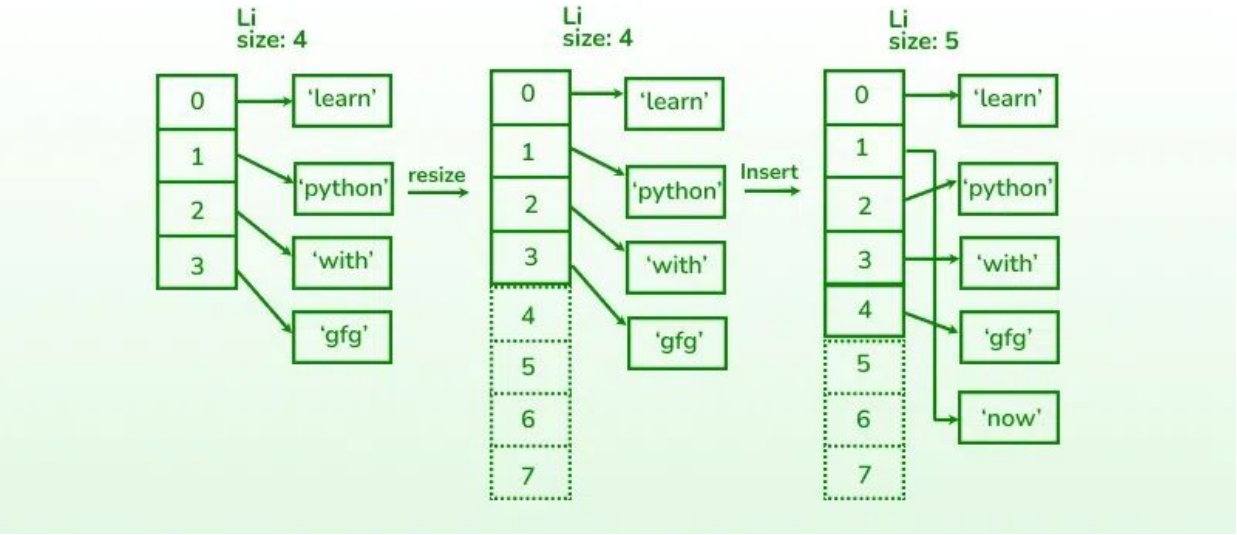
```
0 banana
1 pear
2 orange
```

Internal Representation

- Lists use a concept called over-allocation.
- When a list is created or extended, Python allocates more memory than is needed for the current elements.
- This minimizes the need for frequent resizing during list growth
  - which improves performance but uses more memory.

Lists in memory

- To accommodate future additions, Python pre-allocates extra memory beyond the current size of the list.
- This allocation is beyond the size of the current elements, ensuring efficient resizing for subsequent additions.
  - avoids the need to reallocate memory every time a new element is added.
- When a new element is added, it is inserted into one of the pre-allocated empty slots.



Adapted from: Memory Management in Lists and Tuples using Python - GeeksforGeeks

Exercises 1

- Do these codecheck exercises.

Exercises 2

- Do these codecheck exercises.

Sequence particularly relevant in programming are …

SEQUENCES OF CHARACTERS (STRINGS)

Strings

- Strings are **sequences** of characters.
- String literals are delimited by single or double quotes.

```
fruit = 'orange'
color = "red"
```

- Like other sequences, we can use indexing and slicing.

```
letter = fruit[0]  #-> 'o' (1st character)
fruit[1:4]         #-> 'ran'
fruit[:-1]         #-> 'orang'
fruit[::-1]        #-> 'egnaro'
len(fruit)         #-> 6   (length of string)
```

- We can also concatenate and repeat strings.

```
name = 'tom' + 'cat'   #-> 'tomcat'
gps = 2 * 'tom'        #-> 'tomtom'
```

- For strings, the **in** operator returns True iff the first string appears as a substring in the second.

Strings

- Characters (letters, digits, punctuation) are stored as numeric codes (according to Unicode).
  - `ord(c)` - returns the code of the character c.
  - `chr(n)` - returns the character represented by code n.

- The relational operators work on strings (and other sequences).

```
if word < 'banana':
    print(word, 'comes before banana.')
elif word > 'banana':
    print(word, 'comes after banana.')
else:
    print('the same')
```

Strings are immutable

- Unlike lists, strings in *Python* **are immutable**.
  - Once a string is created it can't be modified.

```
fruit[0] = 'a'          #-> TypeError
```

- But we can create new strings by combining existing ones.

```
ape = fruit[:-1] + 'utan'    #=> ape = 'orangutan'
```

- Methods that imply modification return a new string object.

```
fruit.upper()              #-> 'ORANGE'
fruit.replace('a', 'A')    #-> 'orAnge'
fruit                      #-> 'orange' (not changed)
```

String traversal

- One way to traverse strings is with a for loop:

```
fruit = 'banana'
for char in fruit:
    print(char)
```

```
b
a
n
a
n
a
```

- Another way:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Application example:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Application Examples

- Counting the number of times the letter 'a' appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)  #-> 3
```

- Print common characters in two strings.

```
word1 = 'bananas'
word2 = 'blues'
for letter in word1:
    if letter in word2:
        print(letter)
# outputs b and s
```

Reminder: For strings, the `in` operator returns True if and only if the first string appears as a substring in the second.

String methods

- Strings have various built-in methods, useful for:
  - Checking for different classes of characters
    - `isalpha(), isdigit()` …
  - Case and format
    - `upper(), lower(), title()` …
  - Cleanup and replace
    - `strip(), replace()` …
  - Split and join
    - `split(), join()`

String methods

- Partial list:

Play ▸

`str.isalpha()`	True if all characters are alphabetic.
`str.isdigit()`	True if all characters are digits.
`str.is...`	...
`str.upper()`	Convert to uppercase.
`str.lower()`	Convert to lowercase.
`...`	...
`str.strip()`	Remove leading and trailing whitespace.
`str.lstrip()`	Remove leading whitespace.
`str.rstrip()`	Remove trailing whitespace.
`str.split()`	Split str by the whitespace characters.
`str.split(sep)`	Split str using sep as the delimiter.
`sep.join(lst)`	Join the strings in lst using delimiter sep.

String methods (Python documentation).

strip()

- The `strip()` method is commonly used to remove leading and trailing whitespace (or other specified characters) from strings.
- Example 1 - Cleaning user input:

```
user_input = "   hello world   "
cleaned = user_input.strip()
print(cleaned)  # Output: "hello world"
```

- Example 2 – Removing specific charaters

```
filename = "---report.pdf---"
cleaned_name = filename.strip("-")
print(cleaned_name)  # Output: "report.pdf"
```

replace()

- Typical use case: Modify specific words or phrases in a sentence
  - great for customizing messages or correcting input.

```
text = "Hello, world!"
new_text = text.replace("world", "António")
print(new_text)  # Output: "Hello, António!"
```

- Another example: Clean up formatted strings like IDs, phone numbers, or dates by removing punctuation.

```
cpf = "123.456.789-00"
clean_cpf = cpf.replace(".", "").replace("-", "")
print(clean_cpf)  # Output: "12345678900"
```

split()

- Example: Splitting a Sentence into Words

```
sentence = "Programming is fun to learn"
words = sentence.split()

print("Original sentence:", sentence)
print("Split parts:")
for word in words:
    print("-", word)
```

Original sentence:
Programming is fun to learn
Split parts:
- Programming
- is
- fun
- to
- learn

- Example 2: Splitting by a Specific Character

```
date = "2025-10-13"
parts = date.split("-")

print("Date parts:", parts)
```

Date parts: ['2025', '10', '13']

TUPLES

Tuples

- Contrary to a list, a **tuple** is an immutable sequence of values of any type.
  - The values are indexed by integers, like in lists.
  - The important difference is that tuples are immutable.
- Syntactically, a tuple is a comma-separated list of values.

```
t = 'a', 'b', 'c', 'd', 'e'
```

- It is common (and sometimes necessary) to enclose tuples in parentheses.

```
t = ('a', 'b', 'c', 'd', 'e')
```

- To create a tuple with a single element, you have to include a final comma:

```
t1 = ('a',)
type(t1)    #->  <type 'tuple'>
```

Tuples (2)

- **Another way to create a tuple** is the built-in function **`tuple()`**.

- With no argument, it creates an empty tuple:

```
t = tuple()          # t → ()
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
t = tuple('ape')     # t → ('a', 'p', 'e')
t = tuple([1, 2])    # t → (1, 2)
```

Tuples (3)

- Most list operators also work on tuples.

- We can't modify the elements in a tuple, but we can replace one tuple with another.

```
t = (1, 2)
t = t + (3, 4)      # t → (1, 2, 3, 4)
```

Zipping and Enumerating

- The built-in function `zip()` takes two or more sequences and generates a sequence of tuples
  - each containing one element from each sequence

```
s = 'abc'
t = [4, 3, 2]
list(zip(s, t))     # → [('a', 4), ('b', 3), ('c', 2)]
```

- `enumerate()` generates a sequence of (index, item) pairs.

```
enumerate('abc')    # → (0, 'a'), (1, 'b'), (2, 'c')
```

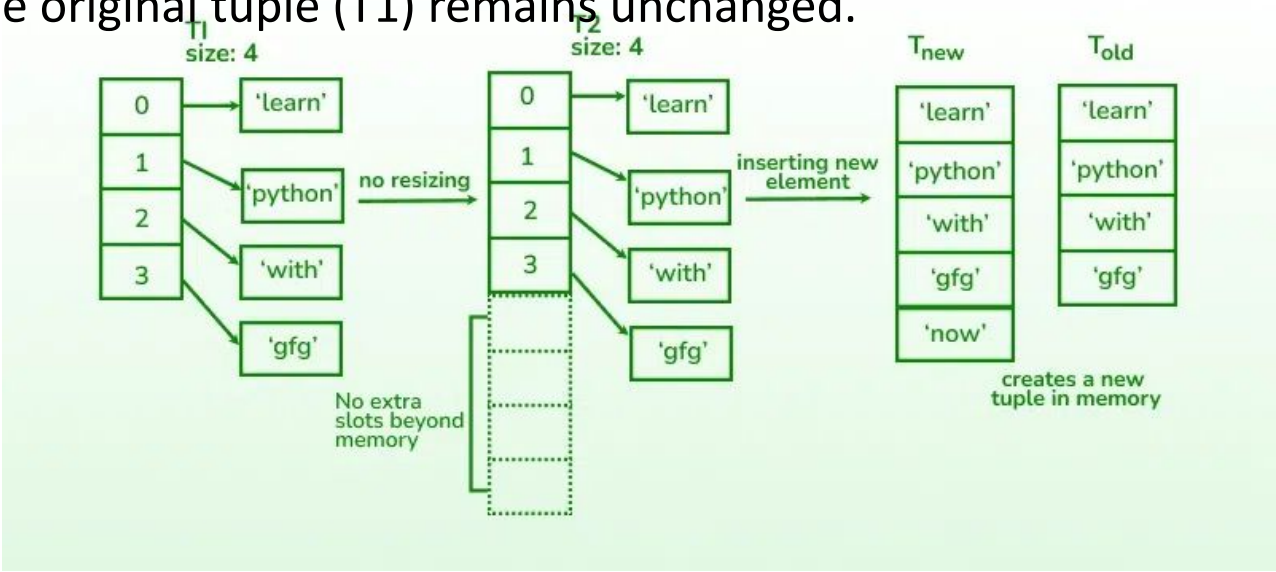
- You can use tuple assignment in a for loop to traverse a sequence of tuples:

```
s = 'somestuff'
for i, c in enumerate(s):
    print(i, c)
```

Play ▸

Tuples in memory

- Memory allocated for the tuple corresponds exactly to its size.
  - No extra memory is allocated for future growth because tuples are immutable.
- When a new element ('now') “is added”, a new tuple (Tnew) is created in memory.
  - This tuple contains all the elements from the old tuple (T1) plus the new element.
  - The original tuple (T1) remains unchanged.



Exercises 3

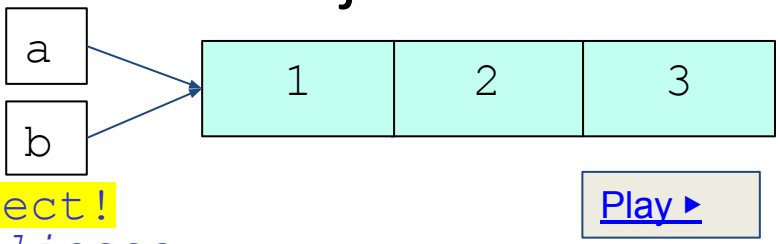
- Do these codecheck exercises.

IMPORTANT CONCEPTS

Aliasing (and Mutability)

- **Aliasing** = Different names referring to the same object.
- In Python, variables store **references** to objects.

```
a = [1, 2, 3]
b = a
# a and b refer to the same object!
# In other words, a and b are aliases.
```



- We can confirm that a and b refer to the same object.

```
a is b          #-> True
```

- If object is changed under one name, it is changed under all names!
  - This may seem strange, at first.

```
b[0] = 9  # object referenced by b is modified
b         #-> [9, 2, 3] of course!
a         #-> [9, 2, 3] even though we did not change a!
```

Aliasing and argument passing

- Aliasing occurs when we pass objects as arguments.
- If the function changes the object, this reflects outside, too.

```
def grow(lst):
    lst.append(3)
    return lst[0] + lst[-1]

lst1 = [1, 2]
x = grow(lst1)
print(x, lst1)   # What's the output?
```

Play ▶

- This **is memory-efficient** and can be very useful.
- But if you don't want it, just make a copy before changing.

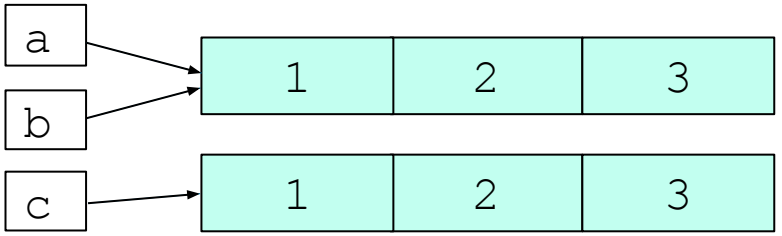
```
def grow(lst):
    r = lst[:]  # makes a copy of lst
    r.append(3)
    return r[0] + r[-1]
```

Equality versus identity

- Objects may be equal without being the same!

```
a = [1, 2, 3]
b = a
c = a[:]
```

Play ▸



Lists c and a are equal, but not the same.

- We test equality with == (or !=).

```
a == b   #-> True        a != b   #-> False
a == c   #-> True        a != c   #-> False
```

- We test identity with **`is`** (or is not).

```
a is b   #-> True        a is not b   #-> False
a is c   #-> False       a is not c   #-> True
```

- Identity implies equality!
- Equality does not imply identity.

Identity and immutable types

- Don’t use `is` when you mean `==` !

```
[1, 2] == [1, 2]     #-> True
[1, 2] is [1, 2]     #-> False

"abx"[:2] == "ab"    #-> True
"abx"[:2] is "ab"    #-> False (probably...)

1000+1 is 1001       #-> False (probably...)
```

- For some immutable types, Python can sometimes detect equal values and share the same object to save space.

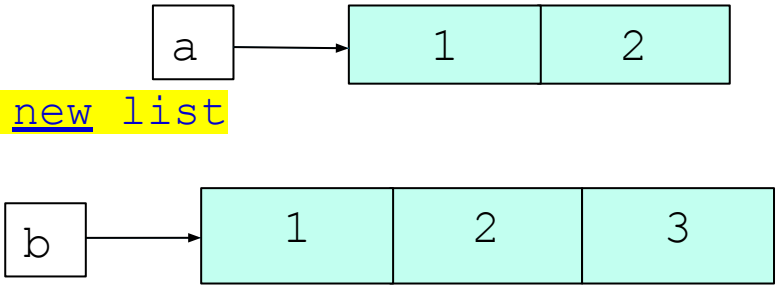
```
"ab" is "ab"     #-> probably True, but ...
10+1 is 11       #-> probably True, but ...
```

- This is implementation-dependent, so do not rely on it!

Making copies / Cloning

- Sometimes, we need to make a copy of an object, so we can change it without changing the original.
- To clone lists, we may use the slicing operator [:].

```
a = [1, 2]
b = a[:]    # slicing creates a new list
b is a  #-> False
b.append(3)
```



- We could also use the more general `copy` method.

```
b = a.copy()   # clone a
b is a         #-> False
```

- Other mutable types (such as sets and dictionaries) also have a copy method.
- Immutable types (tuples, strings) don't need one.
  - Any operation on them creates always a new object

Ordering of sequences

- The **relational operators** work with sequences:
  - Python **starts by comparing the first element from each sequence**.
  - If those elements differ, it returns the result of their comparison.
  - If they are equal, it compares the next elements and repeats.

```
"abacus" < "abel"                  #-> True
(25, 4, 1974) < (25, 12, 1900)     #-> True
"aula9" < "aula10"                 #-> False
```

Play ▸

- The `sorted()` function and the list `sort` method work the same way.
  - They sort primarily by first element, but in the case of a tie, they sort by second element, and so on.

SOME COMMON MISTAKES BEGINNERS MAKE WITH SEQUENCES IN PYTHON

And How to Identify & Fix Them

Conceptual confusions

String vs List Operations

- **Mistake**: Using .append() on a string.

```
s = "hello"
s.append("!")
```

- **Fix**: Use string concatenation: `s += "!"`

Mutability Misunderstanding

- **Mistake**: Trying to modify a tuple element.

```
t = (1, 2, 3)
t[0] = 99
# TypeError: 'tuple' object does not support
  item assignment
```

Indexing and Slicing Error

Off-by-One Indexing

- **Mistake**: Accessing an index that's out of range.

```
lst = [10, 20, 30]
print(lst[3])  # IndexError
```

Incorrect Slice Boundaries

- **Mistake**: Assuming slicing includes the end index.

```
s = "abcdef"
print(s[2:4])  # 'cd', not 'cde'
```

Copying and Identity Pitfall

Confusing Copy with Reference

- **Mistake: Modifying one list affects another**

```
a = [1, 2]
b = a
b.append(3)
print(a)  # [1, 2, 3]
          # but maybe unexpected!
```

- **Fix:** Use b = a.copy() or b = list(a) for a shallow copy.

Type and Equality Confusion

Using `==` vs `is`

- **Mistake**: Checking identity instead of equality.

```
a = [1, 2]
b = [1, 2]
print(a == b)  # True

print(a is b)  # False
               # different objects
```

Iteration Mistake

Modifying a List While Iterating

- **Mistake**: Removing items during iteration.

```
nums = [1, 2, 3, 4]
for n in nums:
    if n % 2 == 0:
        nums.remove(n) # Skips elements
print(nums)  # ✅ [1, 3]
```

- Fix: Iterate over a copy: `for n in nums[:]`

Membership Mistake

Using `in` with Wrong Type

- **Mistake**: Checking for a substring in a list of strings.

```
words = ["apple", "banana"]
print("app" in words) # False
                      # checks full items
```

RETURNING TO OUR INITIAL PROBLEM …

Possible Solution in Python

```
# 1 Define a group of students
students = ["Alice", "Bruno", "Carla", "Diana"]

# 2 For each student in the group:
retake_counts = []
for student in students:
    # 2.1 Ask how many times they retook the quiz;
    count = int(input(f"How many times did {student} retake the quiz? "))

    # 2.2 Store this information in a sequence
    retake_counts.append(count)
```

Possible Solution in Python (cont.)

```
# For each entry in the sequence: Display the student's name and their retake count
print("\nSummary of Retakes:")
for i in range(len(students)):
    print(f"{students[i]} retook the quiz {retake_counts[i]} times.")

# Identify students with high retake counts
# Display their names as needing extra support
print("\nStudents who may need extra help:")
for i in range(len(students)):
    if retake_counts[i] >= 2:
        print("-", students[i])
```