- Comparison operators
- Logical operators
- Flow control
- Loops
- Range
- Enumerator
- Zip
- Random
- And more

TOTAL Python

comparison operators

As their name suggests, they are used to compare two or more values. The result of this comparison is a boolean (True/False)

==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

```
my_bool = 5 >= 6
print(my_bool)
>> False
```

```
my_bool = 5 != 6
print(my_bool)
>> True
```

```
my_bool = 10 == 2*5
print(my_bool)
>> True
```

```
my_bool = 5 < 6
print(my_bool)
>> True
```

```python
a != b
```

(Note: The `<>` operator is recognized as not equal to in Python 2.x versions but is not available in Python 3.x.)

3. Greater than (`>`)

Checks if the value of the left operand is greater than the value of the right operand. If yes, the condition becomes true.

```python
a > b
```

4. Less than (`<`)

Checks if the value of the left operand is less than the value of the right operand. If yes, the condition becomes true.

```python
a < b
```

5. Greater than or equal to (`>=`)

Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, the condition becomes true.

```python
a >= b
```

6. Less than or equal to (`<=`)

Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, the condition becomes true.

```python
a <= b
```

Example Usage:

```python
a = 5
b = 10

# Equal to
print(a == b) # Output: False

# Not equal to
print(a != b) # Output: True

# Greater than
print(a > b)  # Output: False

# Less than
print(a < b)  # Output: True

# Greater than or equal to
print(a >= b) # Output: False

# Less than or equal to
print(a <= b) # Output: True
```

These comparison operators are often used in control flow statements, like `if`, `elif`,

logical operators

TOTAL Python

These operators allow decisions to be made based on multiple conditions.

```
a = 6 > 5
b = 30 == 15*3
```

and returns True if *all conditions* are true

```
my_bool = a and b
print(my_bool)
>> False
```

or returns True if *at least one condition* is true

```
my_bool = a or b
print(my_bool)
>> True
```

not *reverse* the result, returns False if the result is true

```
my_bool = not a
print(my_bool)
>> False
```

Logical operators in Python are used to perform logical operations on the provided operands. They are typically used with Boolean (logical) values, and they return a Boolean value as a result. Python includes three basic logical operators:

1. `and` (Logical AND)

The `and` operator returns `True` if both the operands (expressions) are true. If at least one of the operands is false, the result will be `False`.

Example:

```python
True and True   # Returns: True
True and False  # Returns: False
```

2. `or` (Logical OR)

The `or` operator returns `True` if at least one of the operands (expressions) is true. If both operands are false, the result will be `False`.

Example:

```python
True or True    # Returns: True
True or False   # Returns: True
False or False  # Returns: False
```

3. `not` (Logical NOT)

The `not` operator is used to invert the logical state of its operand. If the operand is `True`, it returns `False`, and if the operand is `False`, it returns `True`.

Example:

3. `not` (Logical NOT)

The `not` operator is used to invert the logical state of its operand. If the operand is `True`, it returns `False`, and if the operand is `False`, it returns `True`.

Example:

```python
not True   # Returns: False
not False  # Returns: True
```

Example Usage:

```python
x = 5
y = 10

# Using 'and' operator
if x == 5 and y == 10:
    print("Both conditions are true")  # Output: Both conditions are true

# Using 'or' operator
if x == 5 or y == 12:
    print("At least one condition is true")  # Output: At least one conditio

# Using 'not' operator
if not x == 7:
    print("x is not equal to 7")  # Output: x is not equal to 7
```

Logical operators can be used to create more complex logical expressions by combining them with comparison operators or other logical expressions. This is particularly useful in control flow statements, like `if`, `elif`, and `else`, to perform different actions based on the evaluation of one or more conditions.

Queue
repl.it

decision making

TOTAL Python

Flow control determines the order in which the code of a program is executed. In Python, the flow is controlled by conditional structures, loops, and functions.

conditional structures (if statement)

Decision making in programming refers to the ability of a program to make decisions based on specified conditions and thereby choosing a specific path of execution. In Python, decision making is accomplished through conditional statements which are primarily the `if`, `elif`, and `else` statements.

1. `if` Statement:

An `if` statement evaluates a condition, and if it is `True`, the block of code within the `if` statement is executed.

Example:

```python
x = 10
if x > 5:
    print("x is greater than 5")
```

2. `elif` Statement:

`elif` (short for else-if) is used for multiple conditions. It comes after an `if` statement and before an `else` statement. The `elif` statement allows you to check multiple expressions for `True` and execute a block of code as soon as one of the conditions is met.

Example:

```python
x = 10
if x > 15:
    print("x is greater than 15")
elif x > 5:
    print("x is greater than 5 but not greater than 15")
```

3. `else` Statement:

`else` comes after `if` and `elif` statements. It catches anything which isn't caught by the preceding conditions.

Example:

```python
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

4. Nested `if` Statements:

You can have `if`/`elif`/`else` statements inside another `if`/`elif`/`else` statements. This is called nested `if` statements.

Example:

```python
x = 10
if x > 5:
    print("x is greater than 5")
    if x == 10:
        print("x is equal to 10")
else:
    print("x is not greater than 5")
```

5. `if`-`else` in One Line (Ternary Operator):

Python allows you to write the `if`-`else` statement in a single line, which is sometimes referred to as the ternary operator.

Example:

```python
x = 10
message = "Even" if x % 2 == 0 else "Odd"
print(message)  # Output: Even
```

Example of Decision Making:

```python
age = 20
if age < 13:
    print("Child")
elif age < 20:
    print("Teenager")
else:
    print("Adult")
```

In this example:

If `age` is less than 13, "Child" is printed.
If `age` is 13 or more but less than 20, "Teenager" is printed.
If `age` is 20 or more, "Adult" is printed.

This demonstrates basic decision-making in Python: depending on the condition met, a specific block of code will be executed.

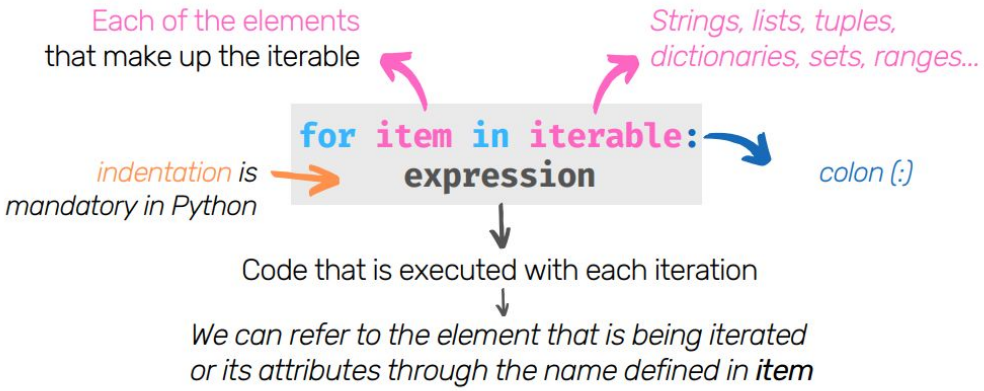
for loops

TOTAL Python

> Unlike other programming languages, for loops in Python have the ability to iterate through the elements of any sequence (lists, strings, etc.), in the order in which those elements are stored.

Concepts

- **Loops:** code sequences that are executed repeatedly
- **Iterable:** object capable of returning its members one at a time, permitting it to be iterated over




```
names = ['John', 'Ann', 'Chad', 'Lee', 'Mary']
```

```
for item in names:
    print("Hello " + item)
```

```
>> "Hello John"
>> "Hello Ann"
>> "Hello Chad"
>> "Hello Lee"
>> "Hello Mary"
```

for_loop.py

```python
my_list = ['Paul', 'Laura', 'Fede', 'Louis', 'Julia']

for name in my_list:
    if name.startswith('L'):
        print(name)
    else:
        print("This name does not begin with 'L")
```

for_loop.py

```python
dic = {'key1':'a', 'key2':'b', 'key3':'c'}

for a,b in dic.items():
    print(a, b)
```

for_loop.py

```python
numbers = [1, 2, 3, 4, 5]
my_value = 0

for number in numbers:
    my_value = my_value + number

print(my_value)
```

In Python, a `for` loop is used for iterating over a sequence (that can be a list, a tuple, a dictionary, a set, or a string) or other iterable objects. Iterating over a sequence is called traversal.

Here's a basic structure of a `for` loop in Python:

```python
for variable in sequence:
    # code to be executed
```

Here `variable` is the variable that takes the value of the item inside the sequence on each iteration. The `sequence` is a collection over which the iteration is to be performed. The code inside the loop is executed once for each item in `sequence`.

Example 1: Loop through a List

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Here, `fruit` is the variable that takes the value of each element in `fruits` list in each iteration.

Example 2: Loop through a String

```python
for char in "banana":
    print(char)
```

In this example, `char` takes on the value of each character in the string "banana" during the loop.

The `range()` Function:

Often, you'll want to iterate over a sequence of numbers. The built-in function `range()` comes in handy in such case. `range()` generates a sequence of numbers, which can be used with a `for` loop.

```python
for i in range(5):  # equivalent to range(0, 5)
    print(i)
```

Looping Through Multiple Lists:

You can loop through multiple lists using the built-in `zip()` function.

```python
list1 = [1, 2, 3]
list2 = ["one", "two", "three"]
for i, j in zip(list1, list2):
    print(i, j)
```

`else` in `for` Loop:

The `else` block just after `for`/`while` is executed only when the loop is NOT terminated by a `break` statement.

```python
for i in range(5):
    print(i)
else:
    print("Loop finished")
```

Nested `for` Loops:

A nested loop is a loop inside a loop. The inner loop will be performed once for each iteration of the outer loop.

```python
for i in range(3):  # outer loop
    for j in range(3):  # inner loop
        print(i, j)
```

`break` and `continue` Statements:

- **`break`**: It terminates the loop containing it and control of the program moves to the next statement following the loop.

```python
for num in range(5):
    if num == 3:
        break
    print(num)
```

- **`continue`**: It is used to skip the rest of the code inside the loop for the current iteration only. The loop does not terminate but continues on with the next iteration.

```python
for num in range(5):
    if num == 3:
        continue
    print(num)
```

These examples cover the basics, but there's a lot more to **`for`** loops, especially when

[Video on loops and for loops]

while loops

TOTAL Python

While loops are another type of loop, but they have more similarities with if conditionals. We can think of while loops as a conditional structure that executes on repeat, until it returns false.

conditional structure

```
while condition:
    expression
else:
    expression
```

colon (:)

indentation is mandatory in Python

This code will be executed when the condition becomes `False`

special instructions

If the code hits a break statement, the loop exits.

The continue statement interrupts the current iteration within the loop, bringing the program to the top of the loop.

The pass statement does not alter the program: it's a placeholder for when a statement is expected, but no action is desired.

while_loop.py

```
coins = 5

while coins > 0:
    print(f"I have {coins} coins")
    coins -= 1
else:
    print("I have no money anymore")
```

while_loop.py

```
answer = 'y'

while answer == 'y':
    answer = input("do you want to continue (y/m)")
else:
    print("Thank you")
```

while_loop.py

```
name = input("Your name")

for letter in name:
    if letter == 'r':
        continue
    print(letter)
```

In Python, the `while` loop is used to iterate over a block of code as long as the test expression (condition) is `true`. Once the condition becomes `false`, the loop stops and control passes to the line immediately following the loop.

Here's the basic syntax of the `while` loop:

```python
while condition:
    # code to be executed
```

Basic Example:

```python
count = 0
while count < 5:
    print(count)
    count += 1  # equivalent to count = count + 1
```

In the above example:

- `count < 5` is the condition to be checked.
- If it is `true`, the block of code inside the `while` loop is executed.
- If it is `false`, the loop terminates, and the control is passed to the next statement after the `while` loop.

`while` Loop with `else`:

Similar to the `for` loop, the `while` loop can also have an optional `else` block which executes after the condition in the `while` statement becomes `false`.

```python
count = 0
while count < 3:
    print(count)
    count += 1
else:
    print("Loop ends.")
```

`break` and `continue` Statements:

- `break`: It is used to exit the `while` loop prematurely when a condition becomes `true`.

```python
count = 0
while count < 5:
    if count == 3:
        break
    print(count)
    count += 1
```

- `continue`: It is used to skip the rest of the code inside the loop for the current iteration and go back to the `while` condition.

```python
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    print(count)
```

Nested `while` Loops:

You can have one **`while`** loop inside another - this is called a nested loop.

python Copy code

```python
count1 = 0
while count1 < 3:
    count2 = 0
    while count2 < 3:
        print(count1, count2)
        count2 += 1
    count1 += 1
```

Infinite `while` Loop:

Be cautious about the condition in the **`while`** loop: if it always evaluates to **`true`**, you will have an infinite loop.

python Copy code

```python
# Infinite loop example (Caution: Do not run this code)
# while True:
#     print("This will print indefinitely")
```

In real-world programming, **`while`** loops are widely used to make programs wait for some external condition to be met, e.g., waiting for user input, waiting for a file to be created, etc. Always ensure the condition in a **`while`** loop will eventually become **`false`**, otherwise, the loop will never terminate!

range( )

TOTAL Python

The range( ) function returns a sequence of numbers given 3 parameters. It is mainly used to control the number of iterations in a for loop.

An integer number specifying at which position to start (**included**)

increment

```
range(start, stop, step)
```

An integer number specifying at which position to stop (**not included**)

```
print(list(range(1,13,3)))
      >> [1,4,7,10]
```

*The only **required** parameter is **stop**.*
The default values for start and step are 0 and 1 respectively.

The `range()` function in Python is a built-in function that generates a sequence of numbers, which is often used for iterating over with loops. `range()` can generate a sequence of numbers from 0 to a specified number, or between two specified numbers, optionally with a specified increment (step). It is commonly used in `for` loops to repeat an action a certain number of times.

The `range()` function has the following syntaxes:

1. `range(stop)`

Generates a sequence of numbers from 0 to `stop - 1`.

2. `range(start, stop)`

Generates a sequence of numbers from `start` to `stop - 1`.

3. `range(start, stop, step)`

Generates a sequence of numbers from `start` to `stop - 1`, incremented by `step`.

Here's a more detailed explanation of the parameters:

- `start` (optional): An integer number specifying at which position to start. Default is 0.
- `stop` (required): An integer number specifying at which position to stop (not inclusive).
- `step` (optional): An integer number specifying the incrementation. Default is 1.

Examples:

Example 1: Using `range(stop)`

python Copy code

```python
for i in range(5):
    print(i)
```

Output:

Copy code

```
0
1
2
3
4
```

Example 2: Using `range(start, stop)`

python Copy code

```python
for i in range(2, 5):
    print(i)
```

Output:

Copy code

```
2
3
4
```

```python
for i in range(0, 10, 2):
    print(i)
```

Output:

```
0
2
4
6
8
```

Example 4: Using `range()` in a List Comprehension

```python
squares = [x*x for x in range(5)]
print(squares)
```

Output:

```csharp
[0, 1, 4, 9, 16]
```

Notes:

- All parameters must be integers, but they may be negative.
- If the `step` value is omitted, it defaults to 1.
- If the `start` value is omitted, it defaults to 0.
- The `stop` value must be provided.
- The values generated by `range()` are not stored in memory until they are needed. The `range()` function creates an object which generates numbers on demand to save memory space.
- To create a list of numbers generated by `range()`, you can use `list(range())`.

Feel free to play around with `range()` in your Python environment to get a feel for how it works!

enumerate( )

TOTAL Python

The enumerate( ) function makes it easy for us to keep track of the iterations, through an iterable index counter, which can be used directly in a loop, or converted to a list of tuples with the list( ) method.

any object that can be iterated over

```
enumerate(iterable, start)
```

defining the start number (int) of the enumerate object (default is 0)

```
print(list(enumerate("Hey!")))
>> [(0, 'H'), (1, 'e'), (2, 'y'), (3, '!')]
```

```
for i, number in enumerate([5.55, 6, 7.50]):
    print(i, number)

>> 0 5.55
>> 1 6
>> 2 7.5
```

In Python, `enumerate()` is a built-in function used to iterate through an iterable (like a list, tuple, string, etc.) in a loop, while also having access to the index of the current item. It returns an enumerate object, which produces pairs containing the index and the item from the original iterable.

Syntax:

```python
enumerate(iterable, start=0)
```

- `iterable`: The iterable to loop through.
- `start` (optional): The value to start counting the index from. Default is 0.

Basic Usage:

```python
fruits = ['apple', 'banana', 'cherry']

for index, item in enumerate(fruits):
    print(index, item)
```

Output:

```
0 apple
1 banana
2 cherry
```

In the example above, `index` takes on the index of `item` within `fruits` during each iteration of the loop.

Using `enumerate()` with `start` Parameter:

You can specify a `start` parameter to change the starting index.

```python
fruits = ['apple', 'banana', 'cherry']

for index, item in enumerate(fruits, start=1):
    print(index, item)
```

Output:

```
1 apple
2 banana
3 cherry
```

Practical Use-Cases:

1. Finding the Index of Certain Items:

Find the indices of all occurrences of a particular item in a list.

```python
letters = ['a', 'b', 'a', 'c', 'd', 'a']
indices = [index for index, letter in enumerate(letters) if letter == 'a']
```

2. Looping Over Two Iterables Together:

Looping through two iterables while keeping track of the index can be done efficiently using `enumerate()` and `zip()` together.

```python
letters = ['a', 'b', 'c']
numbers = [1, 2, 3]

for index, (letter, number) in enumerate(zip(letters, numbers)):
    print(index, letter, number)
```

3. Keeping Line Numbers in Text Processing:

When dealing with text data line-by-line, you might use `enumerate()` to keep track of line numbers.

python Copy code

```python
text = """line1
line2
line3"""

for lineno, line in enumerate(text.splitlines(), start=1):
    print(f"Line {lineno}: {line}")
```

Note:

- Enumerate can be applied to any iterable object.
- It's common to use `_` as a variable name for the index when it is not going to be used in the block.

`enumerate()` can be quite useful in making loops more readable and Pythonic by providing a clean way to access the indices of the items in an iterable while looping through it.

TOTAL Python

zip( )

The zip( ) function acts just like a *zipper*, by creating an iterator formed of the grouped elements of the same index from two or more iterables.

The function stops when the iterable with the fewest elements is exhausted.

```
letters = ['w', 'x', 'c']
numbers = [50, 65, 90, 110, 135]
for letter, num in zip(letters, numbers):
    print(f'Letter: {letter}, and number: {num}')

>> Letter: w, and number: 50
>> Letter: x, and number: 65
>> Letter: c, and number: 90
```

In Python, the `zip()` function is used to combine corresponding elements from multiple iterables (like lists, tuples, etc.) into tuples. The resulting tuples are then yielded one by one as the `zip` object is iterated over. When used with multiple iterables, `zip()` aggregates the i-th element of each iterable into a tuple and returns these tuples in sequence.

Syntax:

python Copy code

```python
zip(iterable1, iterable2, iterable3, ...)
```

Basic Usage:

Example 1: Zipping Two Lists

python Copy code

```python
names = ['Alice', 'Bob', 'Charlie']
ages = [23, 34, 45]

for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

Output:

csharp Copy code

```
Alice is 23 years old
Bob is 34 years old
Charlie is 45 years old
```

Example 2: Zipping Three Lists

python Copy code

```python
names = ['Alice', 'Bob', 'Charlie']
ages = [23, 34, 45]
countries = ['USA', 'Canada', 'UK']

for name, age, country in zip(names, ages, countries):
    print(f"{name}, aged {age}, is from {country}")
```

Output:

csharp Copy code

```
Alice, aged 23, is from USA
Bob, aged 34, is from Canada
Charlie, aged 45, is from UK
```

Key Points:

1. **Equal Length Iterables**: When the iterables are of equal lengths, `zip()` pairs elements based on their order and exhausts when all items are used.
2. **Unequal Length Iterables**: When the iterables are of unequal lengths, `zip()` stops creating tuples when the shortest input iterable is exhausted. If this behavior is not desired, you might look into `itertools.zip_longest()`.

Example with Unequal Lengths:

```python
names = ['Alice', 'Bob', 'Charlie']
ages = [23, 34]

for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

Output:

```csharp
Alice is 23 years old
Bob is 34 years old
```

(Charlie is not included in the output since there's no corresponding age)

1. **Unpacking and Zipping**: You can use `*` (the splat operator) to unzip the data.

Example of Unzipping:

```python
pairs = [('Alice', 23), ('Bob', 34), ('Charlie', 45)]
names, ages = zip(*pairs)
```

Here, `names` will be `('Alice', 'Bob', 'Charlie')` and `ages` will be `(23, 34, 45)`.

Use-Cases:

- **Transposing a Matrix**: If you have a 2D matrix (like a list of lists), you can use `zip()` to transpose rows and columns.
- **Merging Data**: When dealing with multiple related sequences of data, `zip()` can be used to associate the data together.

Convert to Other Types:

To convert the zipped object into a list or some other collection, you can pass it as an argument to the desired type. For example:

```python
zipped_list = list(zip(names, ages))
```

Conclusion:

`zip()` is a flexible and powerful function when dealing with multiple iterables in Python, as it allows you to loop over them in parallel. It aids in writing clean and Pythonic code. Remember to keep in mind the behavior concerning iterables of different lengths, and convert the zipped object to your desired collection type as needed.

min() and max()

TOTAL Python

> The min( ) function returns the item with the lowest value within an iterable. The max( ) function works in the opposite way, returning the highest value of the iterable. If the iterable contains strings, the comparison is done alphabetically.

```
cities_population = {"Albuquerque":559121, "Tulsa":403505}

my_list = [5**5, 12**2, 3050, 475*2]
```

```
print(min(cities_population.keys()))
>> Albuquerque
```

```
print(max(cities_population.values()))
>> 559121
```

```
print(max(my_list))
>> 3125
```

In Python, `min()` and `max()` are built-in functions that return the smallest and largest items, respectively, from an iterable or from two or more arguments. Let's delve deeper into each:

min()

Syntax:

```python
min(iterable, *[, key, default])
min(arg1, arg2, *args[, key])
```

Usage:

1. With a single iterable argument, it returns the smallest item:

```python
print(min([4, 2, 9]))  # Output: 2
```

2. With two or more arguments, it returns the smallest argument:

```python
print(min(4, 2, 9))  # Output: 2
```

3. With a `key` function specified, it returns the item that is smallest according to that function:

```python
print(min(["apple", "banana", "cherry"], key=len))  # Output: apple
```

4. With a `default` value, it returns this default if the iterable is empty:

```python
print(min([], default='No items'))  # Output: No items
```

max()

Syntax:

```python
max(iterable, *[, key, default])
max(arg1, arg2, *args[, key])
```

Usage:

1. With a single iterable argument, it returns the largest item:

```python
print(max([4, 2, 9]))  # Output: 9
```

2. With two or more arguments, it returns the largest argument:

```python
print(max(4, 2, 9))  # Output: 9
```

3. With a `key` function specified, it returns the item that is largest according to that function:

```python
print(max(["apple", "banana", "cherry"], key=len))  # Output: banana
```

4. With a `default` value, it returns this default if the iterable is empty:

```python
print(max([], default='No items'))  # Output: No items
```

Common Applications:

- **Find Extreme Values**: Obviously, to find the minimum or maximum item in a list, tuple, etc.
- **Key-based Extrema**: Identify the smallest or largest item based on a function or property (for example, finding the string with the most characters in a list of strings).
- **Handling Empty Iterables**: With the `default` parameter, you can handle empty iterables gracefully without raising a `ValueError`.

Example:

python Copy code

```python
# Find the longest and shortest word in a list of words
words = ["apple", "banana", "cherry", "dates"]
longest_word = max(words, key=len)
shortest_word = min(words, key=len)

print(f"The longest word is: {longest_word}")
print(f"The shortest word is: {shortest_word}")
```

Output:

arduino Copy code

```
The longest word is: banana
The shortest word is: apple
```

Note:

- `min()` and `max()` each perform a linear search through the data, meaning that they will check every item to determine the minimum or maximum, respectively. Thus, if you're using these functions on large datasets or in a tight loop, keep potential performance issues in mind.

These two functions are simple yet powerful tools in Python for handling a variety of tasks involving comparisons and searching through iterables.

random

TOTAL Python

> Python provides us with a module (a set of functions available for use) that allows us to generate pseudo-random* selections from values or sequences.

Module name

```
from random import *
```

`*` = all methods

can also be independently imported

randint(*min, max*): returns an integer between two given values (both limits included)

uniform(*min, max*): returns a float between a minimum and maximum value

random(): returns a float between 0 and 1

choice(sequence): returns a random element from a sequence of values (lists, tuples, ranges, etc.)

shuffle(*sequence*): takes a mutable sequence of values (like a list), and returns it by randomly changing the order of its elements.

**The mechanics in how these random values are generated is actually predefined in "seeds". While it serves most common uses, it should not be used for security or cryptographic purposes as they can be vulnerable.*

The `random` module in Python provides a variety of functions to generate random numbers and perform random operations. This module uses a pseudorandom number generator, which means the numbers generated are not truly random but are determined by a specific algorithm and initial seed value. However, for most practical purposes, they are sufficiently random.

Here are some useful functions provided by the `random` module:

Basic Random Functions:

random()

Generates a random float number between 0.0 and 1.0.

```python
import random
print(random.random())
```

randint(a, b)

Returns a random integer between `a` and `b` inclusive.

```python
print(random.randint(1, 10))  # Random integer between 1 and 10
```

randrange(start, stop[, step])

Returns a random number from the range specified by `start`, `stop`, and `step`.

```python
print(random.randrange(1, 10, 2))  # Random odd number between 1 and 10
```

Functions for Sequences:

1. **choice(seq)**

   Returns a random element from the non-empty sequence `seq`.

   ```python
   print(random.choice(['apple', 'banana', 'cherry']))  # Randomly select a fr
   ```

2. *choices(population, weights=None, , cum_weights=None, k=1)*

   Returns a `k` sized list of elements chosen from the population with replacement. If a `weights` sequence is specified, selections are made according to the relative weights.

   ```python
   print(random.choices([10, 20, 30, 40, 50], k=3))  # Three selections from t
   ```

3. **shuffle(seq)**

   Shuffles the sequence `seq` in place.

   ```python
   fruits = ['apple', 'banana', 'cherry']
   random.shuffle(fruits)
   print(fruits)  # Fruits are shuffled
   ```

4. **sample(population, k)**

   Returns a new list of length `k` with elements randomly selected from `population`.

   ```python
   print(random.sample(range(1, 100), 5))  # Five unique random numbers from 1
   ```

Real-valued Distributions:

1. **uniform(a, b)**

   Returns a random float number between `a` and `b`.

```python
print(random.uniform(1, 10))  # Random float between 1 and 10
```

2. **gauss(mu, sigma)**

   Returns a random float number based on the Gaussian distribution with mean `mu` and standard deviation `sigma`.

```python
print(random.gauss(0, 1))  # Random float with mean 0 and standard deviatio
```

Usage Notes:

- **Seeding**: You can use `random.seed(a=None, version=2)` to initialize the random number generator. If you use the same seed, you'll get the same sequence of random numbers. This can be useful for debugging or reproducibility.
- **Thread Safety**: The `random` module is not thread-safe without a lock. For simultaneous access by multiple threads, consider using `random.SystemRandom` or manage access using a lock.
- **Cryptographic Uses**: `random` is not suited for cryptography. If you need to generate cryptographically secure random numbers, use the `secrets` module instead.

Here's an example that uses some of the `random` functions:

Here's an example that uses some of the `random` functions:

```python
import random

# Random number between 0 and 1
print(random.random())

# Random integer between 1 and 100
print(random.randint(1, 100))

# Random choice from a list
print(random.choice(['apple', 'banana', 'cherry']))

# Shuffling a list
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)

# Random sample without replacement
print(random.sample(range(100), 5))
```

Remember to `import random` at the start of your script to use these functions. And

TOTAL Python

list comprehension

> List comprehension offers a shorter syntax in creating a new list based on values available in another sequence. It is worth mentioning that brevity comes at the cost of less interpretability.

each element of the iterable

tuples, sets, other lists...

```
new_list= [expression for item in iterable if condition == True]
```

e.g.: mathematic expression

logical operation

Special case with else:

```
new_list= [expression if condition == True else other_expression
                  for item in iterable]
```

Example:

```
new_list = [num**2 for num in range(10) if num < 5]
print(new_list)
>> [0, 1, 4, 9, 16]
```

List comprehensions provide a concise way to create lists in Python. It consists of brackets containing an expression followed by a `for` clause, and optionally, `for` or `if` clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.

Basic Syntax:

```python
[expression for item in iterable]
```

`expression`: The output expression,
`item`: Variable used for each element in the output,
`iterable`: The source of input values.

Example:

```python
squares = [x*x for x in range(10)]
```

Here, `x*x` is the expression, `x` is the item, and `range(10)` is the iterable. The variable `x` takes on the value of each item in the iterable, and the expression evaluates and appends to the new list.

Extended Syntax with `if` Statement:

```python
[expression for item in iterable if condition]
```

Example:

```python
even_squares = [x*x for x in range(10) if x%2 == 0]
```

This will generate a list of squares of even numbers between 0 and 9.

Nested List Comprehensions:

You can nest list comprehensions inside one another.

Example:

```python
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [item for sublist in matrix for item in sublist]
```

Here, the second `for` statement is executed first, followed by the first one.

Practical Applications:

1. Simplifying Loops:

You can replace simple `for` loops that append items to a list with a list comprehension.

```python
# Using a loop
squares = []
for x in range(10):
    squares.append(x*x)

# Using list comprehension
squares = [x*x for x in range(10)]
```

2. Extracting Specific Information:

You can extract specific information from a list of objects.

```python
names = ["Alice", "Bob", "Charlie", "Diana"]
name_lengths = [len(name) for name in names if "a" in name.lower()]
```

This extracts the lengths of names that contain the letter "a".

3. Transposing Matrices:

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed = [[row[i] for row in matrix] for i in range(3)]
```

This swaps rows with columns in the `matrix`.

Noteworthy Points:

- **Readability**: Keep list comprehensions short and simple for readability. If a comprehension gets too complex, it might be better to use a regular loop for the sake of clarity.
- **Performance**: List comprehensions are often faster than equivalent loops, but they can consume more memory in some cases.
- **Nested Loops**: If you have nested loops in a list comprehension, ensure you can read and understand the code easily to prevent bugs and difficulties in debugging.

List comprehensions are a powerful feature in Python, enabling you to write concise and readable code. They are commonly used for creating new lists where each element is the result of some operation applied to each member of another sequence or iterable or to create a subsequence of those elements that satisfy a certain condition.

match

TOTAL Python

> In Python 3.10, structural pattern matching is introduced using the match and case statements. This allows us to associate specific actions based on the patterns of complex data types.

```
match object:
  case <pattern_1>:
    <action_1>
  case <pattern_2>:
    <action_2>
  case <pattern_3>:
    <action_3>
  case _:
    <wildcard_action>
```

The _ character is a wildcard that acts as a match if it does not occur in the previous cases.

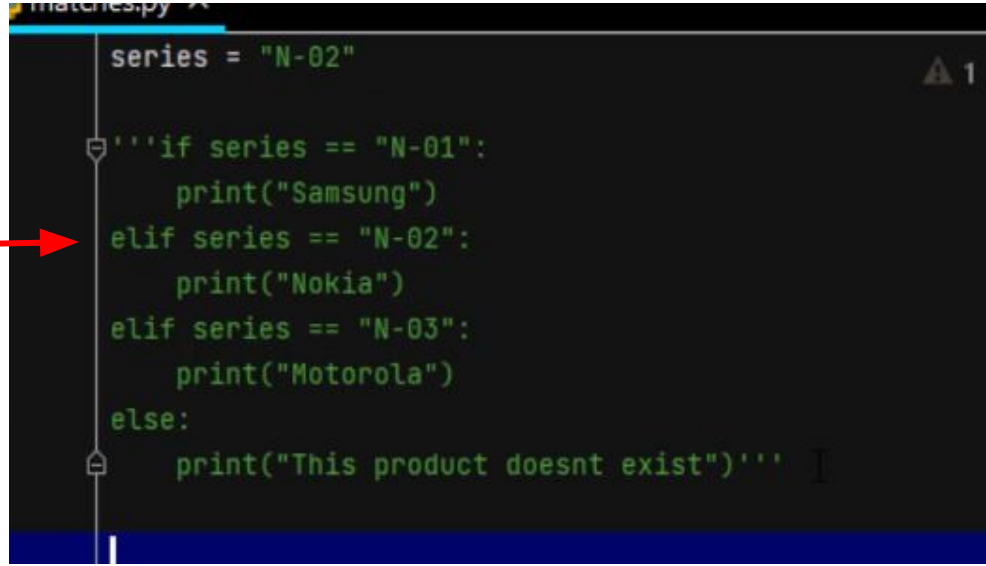
It is possible to detect and deconstruct different data structures: this means that patterns are not only literal values (strings or numbers), but also data structures, on which construction matches are sought.

Match case in python looks for coincidences.. So our code does something different depending on the pattern in the code

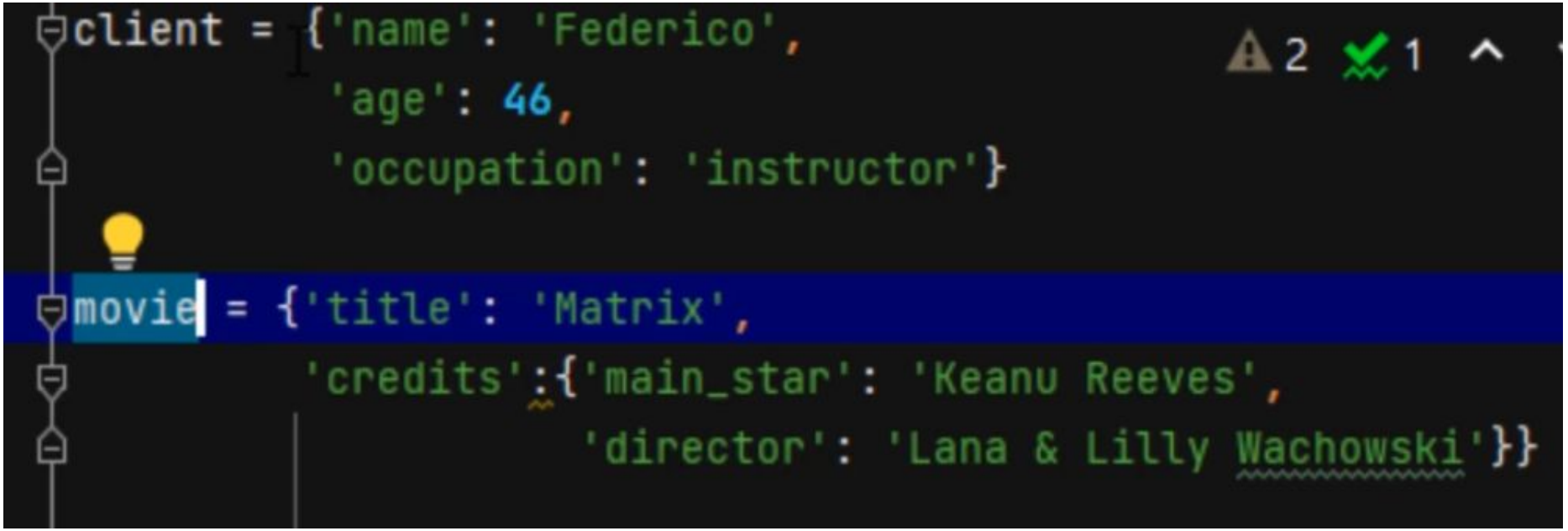
matches.py

```
series = "N-02"

if series == "N-01":
    print("Samsung")
elif series == "N-02":
    print("Nokia")
elif series == "N-03":
    print("Motorola")
else:
    print("This product doesnt exist")
```



```
match series:
    case "N-01":
        print("Samsung")
    case "N-02":
        print("Nokia")
    case "N-03":
        print("Motorola")
    case _:
        print("This product doesnt exist")
```


client = {'name': 'Federico',
'age': 46,
'occupation': 'instructor'}
movie = {'title': 'Matrix',
'credits':{'main_star': 'Keanu Reeves',
'director': 'Lana & Lilly Wachowski'}}
2
1

```python
client = {'name': 'Federico',
          'age': 46,
          'occupation': 'instructor'}

movie = {'title': 'Matrix',
         'credits': {'main_star': 'Keanu Reeves',
                     'director': 'Lana & Lilly Wachowski'}}

items = [client, movie, 'book']


for i in items:
    match i:
        case {'name': name,
              'age': age,
              'occupation': occupation}:
            print("It is a client")
            print(name, age, occupation)
```

```python
movie = {'title': 'Matrix',
         'credits': {'main_star': 'Keanu Reeves',
                     'director': 'Lana & Lilly Wachowski'}}

items = [client, movie, 'book']


for i in items:
    match i:
        case {'name': name,
              'age': age,
              'occupation': occupation}:
            print("It is a client")
            print(name, age, occupation)
        case {'title': title,
              'credits': {'main_star': main_star,
                          'director': director}}:
            print("This is a movie")
            print(title, main_star, director)
        case _:
            print("I don't know what this is")
```

Day 4 Python Challenge

As we learn to program the number of problems we can solve with code also increases. This implies that our challenges are going to be harder to solve. You already know how to use comparison and logical operators, how to control flow and loops. You know super useful statements such as random, zip, range and more...

You are now ready to go up a level and I will ask you to program something a little more complicated than what we have done so far. Today, you are going to create for the first time a fully functional game, with which you can even have some fun yourself.

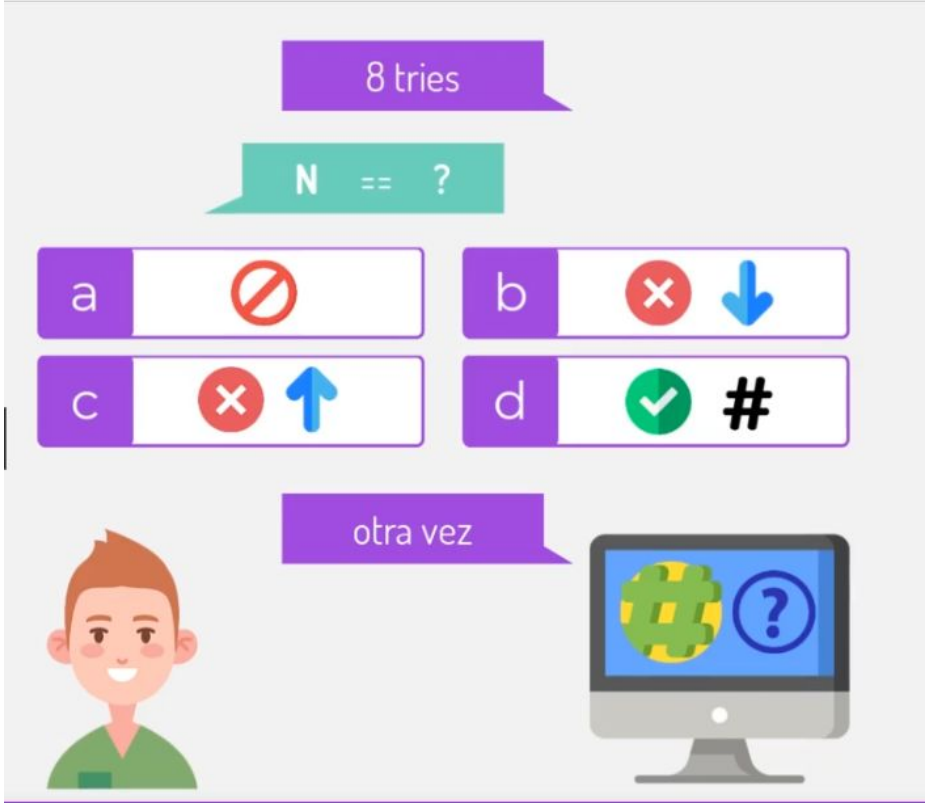
The task is this: the program will ask for the user's name and then it will say something like: "*Well John, I've thought of a number between 1 and 100 and you have only eight tries to guess it. What number do you think it is?*" On each try, the player will say a number and the program can answer for different things.

- **If the number the user said is less than 1 or greater than 100, it will tell them that he/she has chosen a number that is out of play.**
- **If the number the user chose is less than the number the program thought of, it will tell them that the answer is wrong, and that he/she chose a lower number than the secret number.**
- **If the user chose a number greater than the secret number, it will let them know that it was greater.**
- **And if the user got the secret number right, they will be informed that they have won, and how many tries that has taken them.**
- **If the user has not guessed correctly in their first attempt, they will be asked again to choose another number and so on until they win or until their eight attempts are done.**

So, are you up for it? Of course you are.

I remind you the same as always: the most important thing is that you accept this challenge, that you have a good time trying. Try to figure out each step. You know that from what you've learned in this course so far. You have all the knowledge, it's somewhere in there. Maybe you

look at the rubric on the next slide



Requirements breakdown.

Here's a breakdown of how we could solve this challenge:

1. Generate a random number between 1 and 100.
2. Prompt the user for their name and welcome them to the game.
3. Give the user eight tries to guess the number.
4. On each guess, check if the user's input is within the valid range (1-100).
5. If the guess is valid, check if it is less than, greater than, or equal to the secret number.
6. Provide feedback to the user accordingly.
7. Keep track of the number of attempts the user has made.
8. If the user guesses the number correctly within eight attempts, congratulate them.
9. If the user fails to guess correctly after eight attempts, reveal the number.

Evaluation Criteria	Requirements	Points
Functional Requirements (50 points)		
User Input and Interaction (15 points)	Allows user to input name and guesses	5
	Provides appropriate user feedback based on guesses	10
Game Logic (25 points)	Generates and utilizes random secret number	5
	Implements mechanism to check guesses and provide hints	10
	Correctly tracks and limits guesses to eight	10
Win/Lose Conditions (10 points)	Identifies and announces win/lose	5
	Displays number of tries upon winning	5
Code Quality (30 points)		
Readability (10 points)	Uses clear variable names	5
	Includes comments explaining logic	5
Modularity and Functions (10 points)	Breaks program into functions/modules	5
	Each function has single, clear purpose	5
Python Concepts (10 points)		
Comparison Operators	Uses comparison operators	2
Logical Operators	Uses logical operators	2
Flow Control	Implements flow control	1
Loops	Implements loops	1
Range	Uses range()	1
Enumerate	Uses enumerate()	1
Zip	Uses zip()	1
Random	Uses random module	1
User Experience (UX) (10 points)		
User Instructions and Feedback (5 points)	Instructs user how to play	3
	Provides clear feedback after guesses	2
User Engagement (5 points)	Maintains friendly interaction	3
	Congratulates/encourages user appropriately	2
Bonus and Creativity (10 points)		
Additional Features (5 points)	Includes features beyond requirements	5
Code Optimization (5 points)	Implements efficient algorithms/optimizations	5

Total Possible Points: 110

Based on the current rubric, the total possible points are:

- Functional Requirements: 50 points
- Code Quality: 30 points
- Python Concepts: 10 points
- User Experience: 10 points
- Bonus and Creativity: 10 points

So the overall total points for the rubric is **110 points**.

And break your code into **functions** for modularity