

Name: _____
Surname: _____
Date: _____

Intructions

Always assume each code is run independently, so variables do not persist between exercises.

Questions 1-6: Multiple Choice (5 points each)

Circle the correct answer for each question, as each question has **only one** correct answer. A correct answer awards 5 points. There is no penalty for wrong choices.

Questions 7-9: Debugging (10 points each)

These exercises show code cells with one or more errors. You have to find these errors, explain why they happen, and suggest a fix to them.

Questions 10-11: Open Problems (20 points each)

In these exercises, you will encounter problems that are more open-ended and may not have a single correct answer. Your task is to explain how you would approach each problem by breaking it down into smaller, manageable steps.

Describe how you would structure your code, which logical steps are needed, and what kind of Python tools or constructs (e.g., loops, conditionals, functions, lists, etc.) you would use. You are not required to write full code, but doing so is encouraged if it helps clarify your ideas.

1. What will be the final value of the **variable “x”**?

```
1 x = 10 / 4 + 10 // 4 * 2
```

- (a) `x = 4.5`
- (b) `x = 6.5`
- (c) `x = 6.0`
- (d) The code will raise an error.

2. What will happen when we run the following code?

```
1 total = 10
2
3 def add(n: int) -> int:
4     total = total + n
5     return total
6
7 print(add(5))
```

- (a) It prints 15.
- (b) It prints 10.
- (c) The code will raise an error.
- (d) It prints None.

3. After running this code, what will be the final value of the variable “result”?

```
1 word = "python"
2 result = word.upper()[1:4] + word[-1]
```

- (a) result = "YTHn"
- (b) result = "YTHN"
- (c) result = "ythn"
- (d) result = "PYTHn"

4. What will be the output of the following code?

```
1 my_list = [0, 1, 2, 3, 4, 5]
2 my_list += [6, 7, 8, 9]
3 print(my_list[4:])
```

- (a) [4, 5]
- (b) [4, 5, 6, 7, 8, 9]
- (c) [4, 5, [6, 7, 8, 9]]
- (d) Error.

5. Which of the following Python structures is not mutable?

- (a) Lists.
- (b) Tuples.
- (c) Sets.
- (d) Dictionaries.

6. Which one of the following statements about dictionaries is **FALSE**?

- (a) We can initialize an empty dictionary using curly brackets {}.
- (b) Keys in a dictionary must be unique.
- (c) Dictionary keys can only be strings.
- (d) A dictionary can contain other dictionaries inside.

Answers: 1-b, 2-c, 3-a, 4-b, 5-b, 6-c

7. The function below should add the digits of a positive integer. However, when we call it with 123 the code never finishes. Identify and fix the issue(s).

```
1 def digit_sum(n: int) -> int:
2     total = 0
3     while n > 0:
4         total += n % 10
5         n / 10
6     return total
7
8 print(digit_sum(123)) # Expected: 6
9 # Output: the code never stops
```

Answer:

The problem is line 5, `n / 10`. This line computes a division, but it does not store the result anywhere, so the value of `n` never changes. The condition `n > 0` is always true, and the loop never ends. (+5 points)

We need to update the variable, and we have to use the integer division, `n = n // 10`. The goal is to remove the last digit in each iteration (123 becomes 12, then 1, then 0). With the normal division, `n = n / 10`, we would get a float (12.3, 1.23...) that keeps the decimals and does not reach 0 in a reasonable number of steps. (+5 points)

(Optional, fixed code)

```
1 def digit_sum(n: int) -> int:
2     total = 0
3     while n > 0:
4         total += n % 10
5         n = n // 10
6     return total
```

8. The code below should apply a percentage discount to a price. However, it raises an `UnboundLocalError` on line 4. After we fix that error, the code runs, but it prints 100 instead of the expected 80.0. Identify and fix both problems.

```
1 price = 100
2
3 def apply_discount(percent: float) -> None:
4     price = price * (1 - percent / 100)
5
6 apply_discount(20)
7 print(price) # Expected: 80.0
```

Answer:

The first problem is the scope of `price`. Because line 4 assigns a value to `price` inside the function, Python treats `price` as a **local** variable in the whole function. Then, when the right side of the line tries to read it, the local variable has no value yet, and Python raises the error. The global variable `price` is not used. (+5 points)

The second problem is that, even if the function ran, it would only change its own local variable. The global `price` stays at 100, as a function does not modify a global variable by assigning to it. In addition, the function does not return anything. The right fix is to pass the price as an argument, **return** the new price, and store the result outside: `price = apply_discount(price, 20)`. (+5 points)

Comment: Using the keyword `global` inside the function would also work, but it is considered a bad practice. Functions should receive their data as arguments and send their results with `return`.

(Optional, fixed code)

```
1 price = 100
2
3 def apply_discount(price: float, percent: float) -> float:
4     return price * (1 - percent / 100)
5
6 price = apply_discount(price, 20)
7 print(price) # 80.0
```

9. The function below should return the first character that appears twice in a text, reading from left to right, or `None` if no character is repeated. However, it always returns `None`. Identify and fix the issue(s).

```
1 def first_repeated(text: str) -> str:
2     seen = set()
3     for char in text:
4         if char in seen:
5             seen.add(char)
6             return char
7     return None
8
9 print(first_repeated("abcba")) # Expected: "b"
10 # Output: None
```

Answer:

The problem is the position of line 5, `seen.add(char)`. It is inside the `if`-clause, so a character is only added to the set when it is already in the set, which never happens. The set `seen` stays empty, the condition is always false, and the loop reaches the final `return None`. (+5 points)

We need to add every character to the set, so the line must run in every iteration, after the check. We can move it outside the `if`-clause (at the same indentation as the `if`), or put it inside an `else`. The order matters: if we add the character *before* the check, the condition is always true and the function returns the first character of the text. (+5 points)

(Optional, fixed code)

```
1 def first_repeated(text: str) -> str:
2     seen = set()
3     for char in text:
4         if char in seen:
5             return char
6         seen.add(char)
7     return None
```

10. A library registers its loans as a list of tuples (**book**, **member**), one tuple for each loan. The same member can borrow the same book several times, for instance:

```
1 loans = [("Dune", "Ana"), ("Emma", "Ben"), ("Dune", "Ana"),
2          ("Ulysses", "Ben"), ("Dune", "Cy")]
```

Design a function that, given **any** list of loans, returns two things: (1) for each member, the books that this member has borrowed, without repetitions, and (2) the name of the member who has borrowed the most **different** books. In case of a tie, return the name that comes first in alphabetical order.

Answer:

The input of the function is a list of tuples of two strings. We need to return two things, so the function will return a tuple with two elements: the first one is a dictionary, and the second one is a string with the name of the best member. (+2 points)

First, we initialize an empty dictionary, `books = {}`. We loop through the loans, and we unpack each tuple directly in the loop, `for book, member in loans`. (+4 points)

The keys of the dictionary are the member names, and the values are **sets** of books. A set is the right choice because it ignores repetitions automatically. Inside the loop, if the member is not yet a key (`member not in books`), we create the key with an empty set, `books[member] = set()`. After that, in all cases, we add the book to their set, `books[member].add(book)`. (+6 points)

Next, we look for the best member. We initialize `best_name = None` and `best_count = 0`. We loop through the names **in alphabetical order**, `for name in sorted(books.keys())`. If the number of books of a member, `len(books[name])`, is **strictly greater** than `best_count`, we replace both variables. As we only replace when the number is strictly greater, in case of a tie we keep the member that came first alphabetically. (+5 points)

Finally, we return the tuple (`books, best_name`). (+3 points)

(Optional, code)

```
1 def exercise10(loans: list[tuple]) -> tuple:
2     # Books borrowed by each member, without repetitions
3     books = {}
4     for book, member in loans:
5         if member not in books:
6             books[member] = set()
7             books[member].add(book)
8
9     # Member with the most different books
10    best_name = None
11    best_count = 0
12    for name in sorted(books.keys()):
13        # Strictly greater: ties keep the first name alphabetically
14        if len(books[name]) > best_count:
15            best_name = name
16            best_count = len(books[name])
17
18    return books, best_name
```

11. Run-length encoding is a simple way to compress a text. It replaces every group of consecutive repeated characters with the character followed by the number of repetitions. For instance:

```
1 "aaabccdd" -> "a3b1c2d2"
2 "abc"      -> "a1b1c1"
3 "aabaa"    -> "a2b1a2"    # The two groups of "a" are separate
4 ""         -> ""          # Empty text
```

Design a function that, given **any** string, returns its run-length encoding.

Answer:

The input of the function is a string, and the output is also a string. (+2 points)

We start with the special case: if the text is empty, we return an empty string right away, as there is no first character to look at. (+2 points)

Next, we need to remember which group we are counting. We initialize three variables: `result = ""`, where we build the output, `current = text[0]`, the character of the current group, and `count = 1`, the repetitions seen so far. (+4 points)

Then we loop through the rest of the characters, skipping the first one, which is already in `current`: `for char in text[1:]`. Inside the loop, we use an if-else. If `char == current`, the group continues, so `count += 1`. Otherwise, the group has ended: we add it to the result with `result += f"{current}{count}"`, and we start a new group with `current = char` and `count = 1`. (+8 points)

After the loop ends, the last group has not been added yet, because we only add a group when we find a different character. A very common mistake is to forget it. We add it with the same line, `result += f"{current}{count}"`, and we return `result`. (+4 points)

(Optional, code)

```
1 def exercisel1(text: str) -> str:
2     # Empty text: nothing to encode
3     if len(text) == 0:
4         return ""
5     result = ""
6     current = text[0]
7     count = 1
8     # Look at the rest of the characters
9     for char in text[1:]:
10        if char == current:
11            # The group continues
12            count += 1
13        else:
14            # The group has ended: save it and start a new one
15            result += f"{current}{count}"
16            current = char
17            count = 1
18    # Do not forget the last group
19    result += f"{current}{count}"
20    return result
```

Cheatsheet

Given a number named **x**:

- `x % y` returns the remainder of the division $\frac{x}{y}$.
- `x // y` returns the quotient of the division $\frac{x}{y}$.
- `abs(x)` returns the absolute value of **x**.

Given a string named **s**:

- `int(s)` returns an integer. Fails if **s** has invalid characters.
- `float(s)` returns a floating number. Fails if **s** has invalid characters.
- `s.split(" ")` split a string into a list where each word is a list item.
- `s.count(<pattern>)` returns the number of non-overlapping occurrences of `<pattern>`.
- `s.find(<pattern>, <start>)` finds the first occurrence of the specified `<pattern>`, starting from index `<start>`. Returns `-1` if the pattern is not found.

Given a list or tuple named **ls**:

- `len(ls)` returns the number of elements in that list.
- `sum(ls)` returns the sum of the elements of that list.
- `sorted(ls)` returns a sorted version of that list.
- `tuple(ls) / set(ls)` returns a tuple / set using the elements of that list.
- `", ".join(ls)` returns a string with the elements of the list separated by commas.

Given a set named **a**:

- `a.add(x)` adds the element **x** to the set **a**.
- `a.union(b)` returns the union of set **a** with another set **b**.
- `a.intersection(b)` returns the intersection of set **a** with another set **b**.
- `a.difference(b)` returns the difference of set **a** with another set **b**.

Given a dictionary named **d**:

- `list(d.keys())` returns a list with the keys of **d**.
- `list(d.values())` returns a list with the values of **d**.
- `list(d.items())` returns a list with the tuples (key, value) of **d**.

Some useful build-in functions:

- `type(x)` returns the type of **x** (int, float, list...).
- `range(start, stop)` generates numbers from **start** until **stop**, not including **stop**.