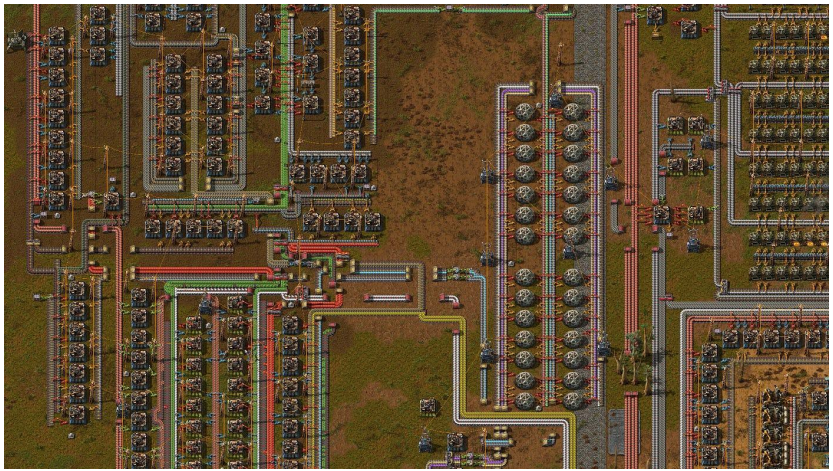
Advanced Control Flow

Agenda - Schedule

1. Warm-Up
2. Conditional Patterns
3. Loops
4. Loop Patterns
5. VSCode Lab



Data engineering is the art of ingesting data without error.

Agenda - Goals

- Understand common logical patterns in Python
- Understand how to implement for loops
- Integrate for-loops with data structures

Warm-Up

```
def sleep_classifier(length: float, avg_bpm: int) -> str:
        if length > 14 or avg_bpm > 100:
                return "right-fence outliers"
        elif length < 3 or avg_bpm < 30:
                return  "left-fence outliers"

        if length >= 6 and avg_bpm <= 70:
                if avg_bpm >= 45 and avg_bpm <= 55:
                        return  "excellent"
                elif avg_bpm > 55 and avg_bpm <= 65:
                        return  "good"
                else:
                        return  "fair"
        else:
                if length < 6 and length >= 5:
                        return  "poor"
                else:
                        return  "abysmal"
        return "unknown"

print(sleep_classifier(6.5, 66))
print(sleep_classifier(2.3, 59))
print(sleep_classifier(4.5, 61))
```

Join your pod groups and evaluate this chunk of code. Work together to figure out what will occur when we run this code.

```
def sleep_classifier(length: float, avg_bpm: int) -> str:
        if length > 14 or avg_bpm > 100:
                return “right-fence outliers”
        elif length < 3 or avg_bpm < 30:
                return  “left-fence outliers”

        if length >= 6 and avg_bpm <= 70:
                if avg_bpm >= 45 and avg_bpm <= 55:
                        return  “excellent”
                elif avg_bpm > 55 and avg_bpm <= 65:
                        return  “good”
                else:
                        return  “fair”
        else:
                if length < 6 and length >= 5:
                        return  “poor”
                else:
                        return  “abysmal”
        return “unknown”

print(sleep_classifier(6.5, 66))
print(sleep_classifier(2.3, 59))
print(sleep_classifier(4.5, 61))
```

Is it possible for this return expression to be evaluated? If so, which value will accomplish this?

You are a data analyst for a local state gov't.

You are tasked with reporting the outcome of a new piece of legislation that provides decreased taxes for businesses that open and hire at least 100 new positions within certain towns that have experienced population loss.

4 years have passed since this tax break went into effect, and your team is tasked with analyzing its outcome on population growth. You calculate differences of 2020 and 2024 population in counties whose businesses participated in this measure.

When calculating the descriptive statistics of this dataset, you discover that median == mode, while median < mean. What does this indicate about the distribution of outcomes?

Culture Contract

Culture Contract

Before we continue with DS technical content, we just want to briefly direct your attention to the Culture Contract that we're operating off of within this fellowship.

A major part of your success in this fellowship and your professional career hinges on your ability to be a **good team player**.

We want you to think of us as your coworkers. Therefore we will hold you to the **same standards as we hold our coworkers and ourselves.**

Culture Contract

Before tomorrow, (if not today) take 10 minutes to read the Culture Contract and consider its implications for what communication looks like for us to you, you to us, and you to your peers.

The successful technical workspace **is as much a social enterprise** as it is an intellectual one.

Culture Contract - Core Principles

We ask for you to abide by the following principles:

- Be **approachable**, **kind**, and **resolution-oriented**
- Develop your **capability to be an authority** in the knowledge you hold
- Be driven by **curiosity**

Any behavior that contradicts this will be pointed out. Continued disregard for **chiefly the first principle** can result in dismissal.

Culture Contract - Example 1

You are working with disagrees with your analysis plan and proposes an alternative. Do you…

A) Immediately call for their dismissal from the team/program
B) Ignore them and continue with your plan
C) Challenge them to duel by virtual combat
D) Consider and discuss the reasons one plan should be preferred over another

Remember, you are not your code. You are not your projects, you are you.

A rejection of your thought-stuff is not a rejection of your character.

Culture Contract - Example 2

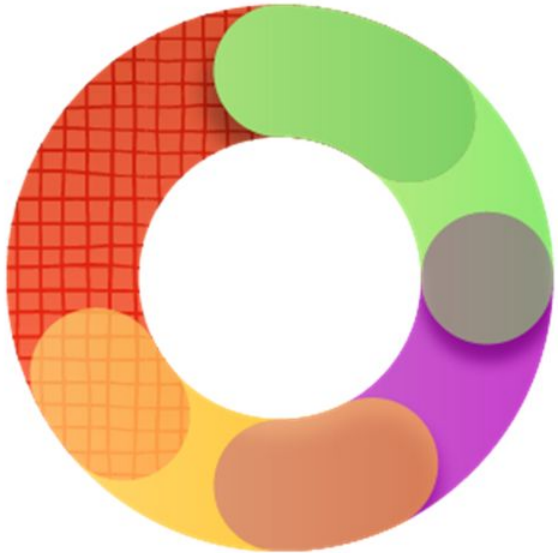
You are working with disagrees with your analysis plan and proposes an alternative. Do you…

A) Immediately call for their dismissal from the team/program
B) Ignore them and continue with your plan
C) Challenge them to duel by virtual combat
D) Consider and discuss the reasons one plan should be preferred over another

Logical Patterns

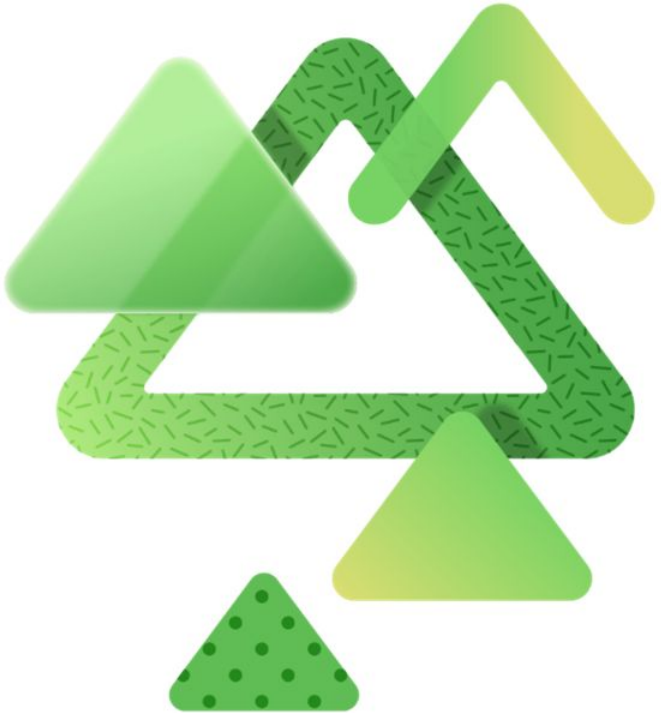
Logical Patterns

- Example patterns to help guide your thinking about using `if` statements in different ways:
  - `and` vs. nested `if`
  - Defensive Guard
  - Early Return
  - Multiple Return Spots
  - Build-Up-Return
  - Define-and-Refine



Logical Patterns

- The patterns:
  - Are not officially part of the Python language.
  - Are templates created based on common patterns seen in code.
  - Are only starting points, because there are many ways to leverage and twist conditionals.



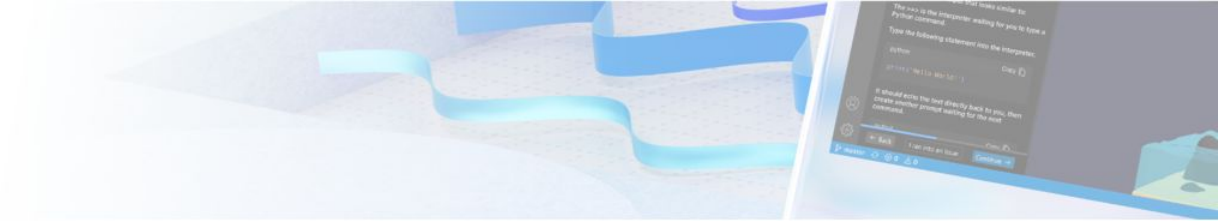
And vs. Nested If

- An `if` statement nested inside an `if` statement is the same as both conditions combined with an `and` statement.
- The advantage of an `and` expression is brevity because the code only takes up a single line without an extra indentation.
- You can add statements before or after the inner `if` statement, allowing more statements for the first case.

And vs. Nested If

- You can specify additional cases' behavior by adding `elif` and `else` statements into the inner `if` statements.
- You can add in these statements if you need to narrow the behavior in the case where the first condition is true and the second condition is false.

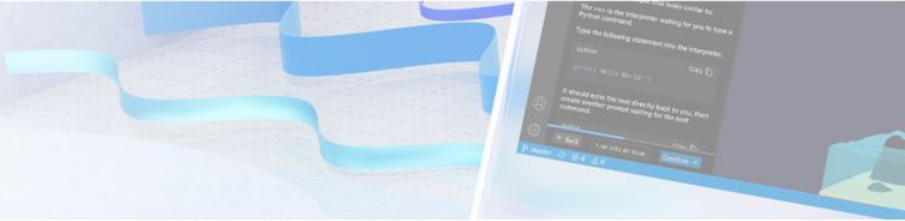
And vs. Nested If



- **Example:** The first version must have redundant versions of the `weather.lower()` check.
- The second version has an extra line of code because of the second `if` statement.
- With less redundant logic, the second block of code is less likely to have typos.

```
if weather.lower() == 'sunny' and temperature > 70:
    print("Warm!")
if weather.lower() == 'sunny' and temperature < 30:
    print("Cold!")
# Is the same as...
if weather.lower() == 'sunny':
    if temperature > 70:
        print("Warm!")
    if temperature < 30:
        print("Cold!")
```

Check Your Understanding

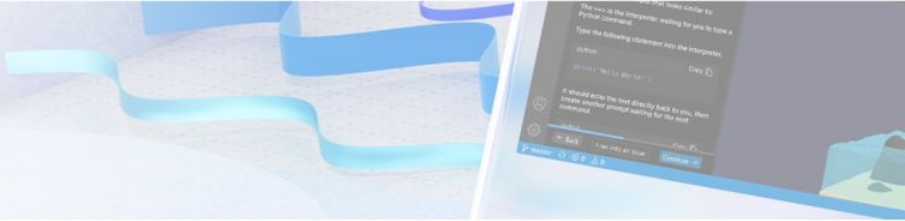


Question 1

What happens when there are multiple return statements in a function?

- ❑ The `return` statements must be placed under other control structures like `if` statements, or the other `return` statements won't be reachable.
- ❑ The function ends when the first `return` statement is reached.
- ❑ The last `return` statement is used to determine the returned value.
- ❑ An error occurs because you can't have more than one `return` statement.

Check Your Understanding



Question 1

What happens when there are multiple return statements in a function?

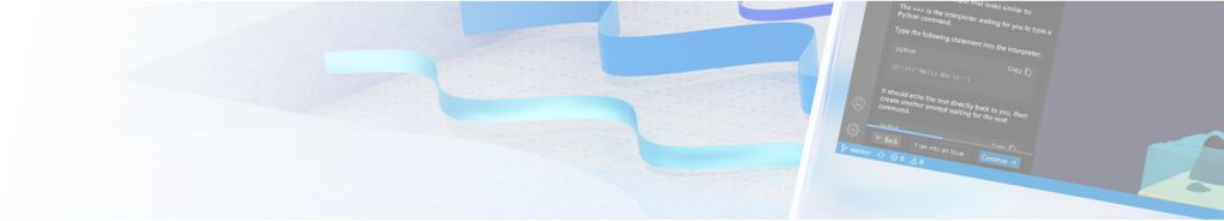
- ✓ The `return` statements must be placed under other control structures like `if` statements, or the other `return` statements won't be reachable.
  Each `return` statement must be guarded. Otherwise, only the first `return` statement will have any effect on the flow of execution.

- ✓ The function ends when the first `return` statement is reached.
  A `return` statement ends the function in Python, regardless of whether there are more lines of code after the `return` statement in the body.

- ❑ The last `return` statement is used to determine the returned value.

- ❑ An error occurs because you can't have more than one `return` statement.

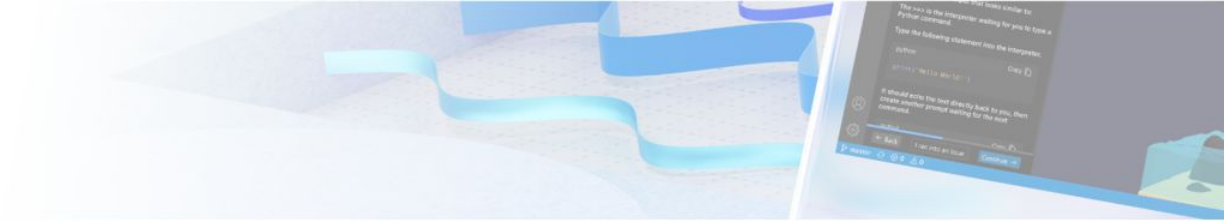
Defensive Guard



The Defensive Guard pattern involves "protecting" operations from errors by first checking whether the data involved is valid.

```
if ___:
# Potentially unsafe operation
```

Defensive Guard

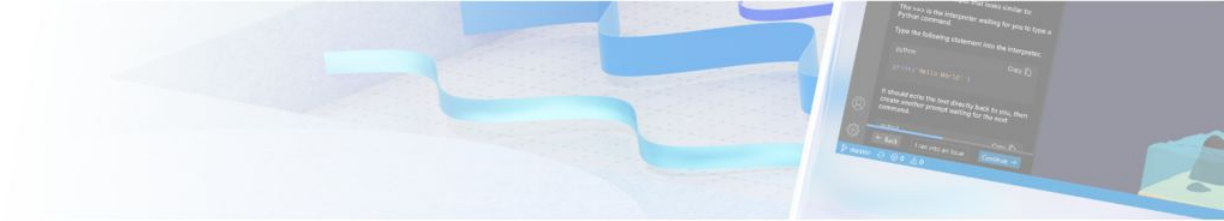


- **Example**: Division by zero causes a `ZeroDivisionError`.

- You can use an `if` statement to ensure sure that a variable such as `time` is greater than zero and safe to use for the denominator.

```
# Guard against division by zero
if time > 0:
    speed = distance / time
```

Defensive Guard

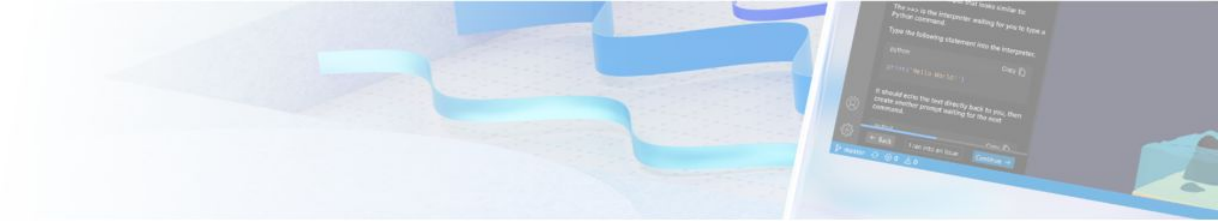


- You can use the `isdigits` method of strings to make sure that the string is all numbers.

- Do this before you try convert the string to a float or an integer.

```
# Guard against bad conversion
if user_input.isdigits():
    age = int(user_input)
```

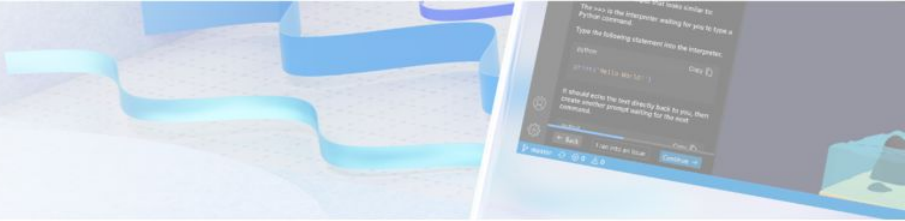
Defensive Guard



- Test the length of a string before you index a specific character.
- You can determine that the string isn't empty by using *truthiness*.
- If you want a specific inner character, you must check that the string is at least that long.

```
# Truthiness to test if there are ANY characters
if user_input:
    last_character = user_input[-1]
    first_character = user_input[0]
# Check length to make sure that there are at least three characters
if len(user_input) >= 3:
    third_character = user_input[2]
```

Check Your Understanding

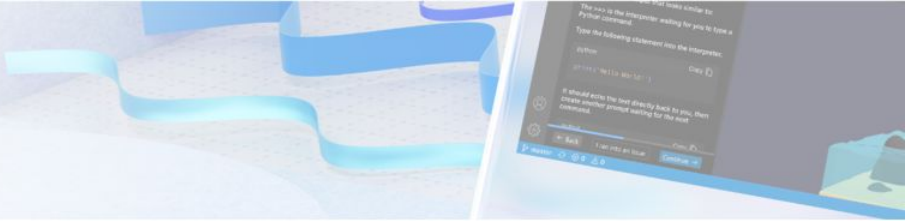


Question 2

In which of these cases might you need a Defensive Guard to avoid an error?

- ❑ Make sure an integer is positive before you convert it to a string.
- ❑ Make sure that a string has at least one character before you index that first character.
- ❑ Make sure that an integer is not zero before a division.
- ❑ Make sure that a string has only digits before you convert the string to an integer.

Check Your Understanding

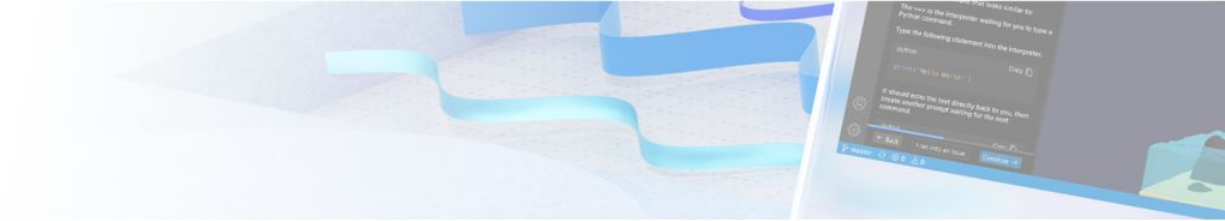


Question 2

In which of these cases might you need a Defensive Guard to avoid an error?

- ❑ Make sure an integer is positive before you convert it to a string.
- ✓ Make sure that a string has at least one character before you index that first character.
  That would cause an `IndexError` otherwise!
- ✓ Make sure that an integer is not zero before a division.
  That would cause a `ZeroDivisionError` otherwise!
- ✓ Make sure that a string has only digits before you convert the string to an integer.
  That would cause a `ValueError` otherwise!

Early Return



The Early Return pattern uses an `if` statement to end a path inside of a function before an error can occur.

```
# Early Return
def function(___):
    if ___:
        return ___
    # do something ...
    return ___
```

Early Return

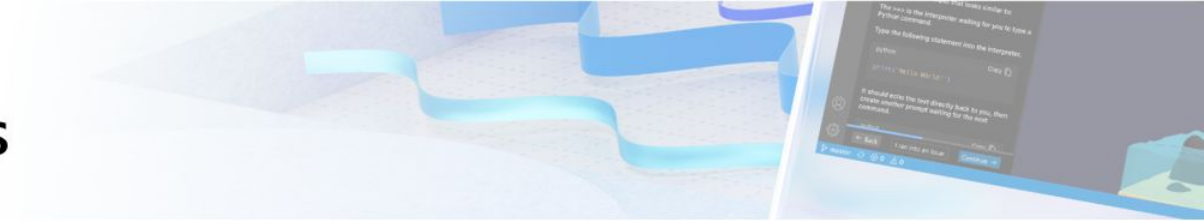
- **Example**: Check whether the `message` parameter has a truthy value.
- Any non-empty string is considered truthy. In this case, indexing is attempted only if the `message` isn't empty.
- Indexing the last character of the string causes an `IndexError` if the empty string is indexed.
- The early `return` prevents the mistake from occurring.

```
def get_punctuation(message: str) -> str:
    if not message:
        return "empty string"
    last_character = message[-1]
    if last_character == "?":
        return "question"
    elif last_character == ".":
        return "statement"
    return "other"
```

Multiple Return Spots

- When parsing code with multiple returns, remember that *a function can return exactly once*.
- If a `return` statement is encountered, the function is over. Whatever value is specified in the `return` statement is what is brought back to the original call site.
- Use more than one `return` statement only when you have `if` statements.
- The `if` statements disrupt the function's control flow, branching off different paths for each `return` statement.
- For each call to the function, only one path needs to be taken, with only one `return` statement activated.

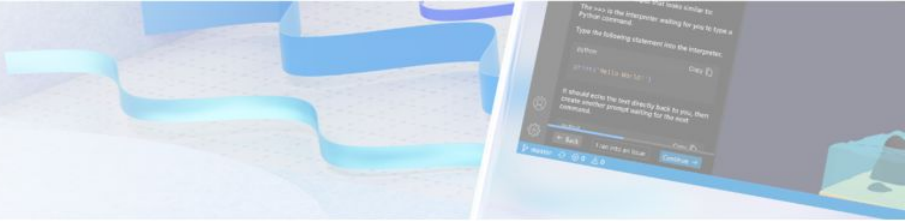
Multiple Return Spots



- The `convert_ordinal` function consumes an integer and produces a string representation of the ordinal number.
- The value stored in `number` is matched against each possibility, but only one can be true.
- If none of the expressions evaluate to `True`, the `else` block is entered and the value `"other"` is returned.

```
def convert_ordinal(number: int) -> str:
    if number == 1:
        return "first"
    elif number == 2:
        return "second"
    elif number == 3:
        return "third"
    else:
        return "other"
```

Check Your Understanding

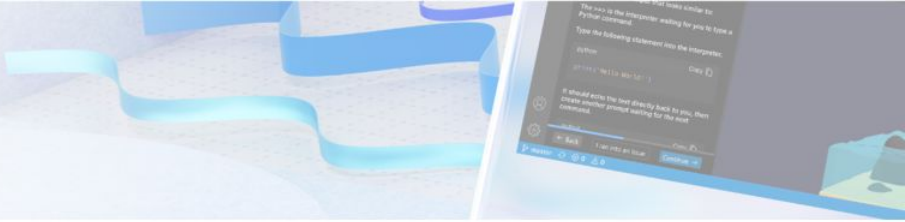


Question 3

What happens when there are multiple `return` statements in a function?

- ❑ An error occurs because you can't have more than one `return` statement.
- ❑ The function ends when the first `return` statement is reached.
- ❑ The last `return` statement is used to determine the returned value.
- ❑ The `return` statements must be placed under other control structures like `if` statements, or the other `return` statements won't be reachable.

Check Your Understanding

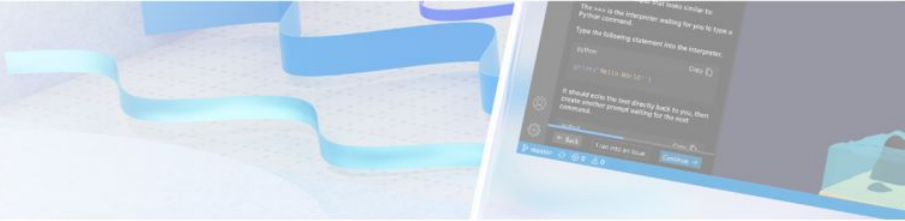


Question 3

What happens when there are multiple `return` statements in a function?

- ❑ An error occurs because you can't have more than one `return` statement.
- ✓ The function ends when the first `return` statement is reached.
  A `return` statement ends the function in Python, regardless of whether there are more lines of code after the `return` statement in the body.
- ❑ The last `return` statement is used to determine the returned value.
- ✓ The `return` statements must be placed under other control structures like `if` statements, or the other `return` statements won't be reachable.
  Each `return` statement must be guarded. Otherwise, only the first `return` statement will have any effect on the flow of execution.

Build-Up-Return Pattern



```
def function(___):
    if ___:
        result = ___
    elif ___:
        result = ___
    # ...
    else:
        result = ___
    return result
```

- In the Build-Up-Return pattern, the value being returned is meant to be the same.
- For flow control, avoid mixing a branch of `if` statements with `return` statements.
- The Build-Up-Return pattern helps you isolate the `return` to a single spot, making it easier to think about.

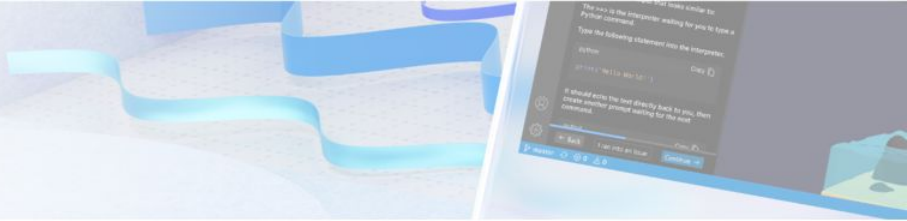
Built-Up-Return Pattern

- The previous `convert_ordinal` pattern had `return` statements inside the `if` statements. Here there's only one `return` statement after all the conditionals.

- Instead, assignment statements are used. The result is the same.

```
def convert_ordinal(number: int) -> str:
    if number == 1:
        result = "first"
    elif number == 2:
        result = "second"
    elif number == 3:
        result = "third"
    else:
        result = "other"
    return result
```

Check Your Understanding

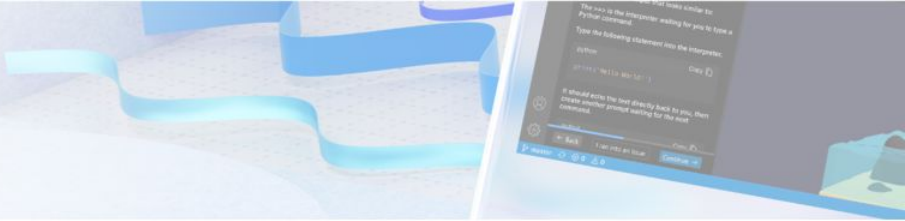


Question 4

Why might you use the Build-Up-Return pattern instead of the Multiple-Returns pattern?

- ❑ The Build-Up-Return pattern guarantees that a value is always returned, unlike the Multiple-Returns pattern where you might forget to return a value from a branch.
- ❑ The Multiple-Returns pattern often leads to shorter code.
- ❑ The Multiple-Returns pattern often leads to faster code.
- ❑ The Build-Up-Return pattern only has one `return` statement, so the control flow is less complicated.

Check Your Understanding



Question 4

Why might you use the Build-Up-Return pattern instead of the Multiple-Returns pattern?

- ✓ The Build-Up-Return pattern guarantees that a value is always returned, unlike the Multiple-Returns where you might forget to return a value from a branch.
  The Build-Up-Return pattern makes it impossible to forget to provide a value, but Multiple-Returns makes it easy. Of course, you should be careful to always unit test your code so that you aren't forgetting things either way!

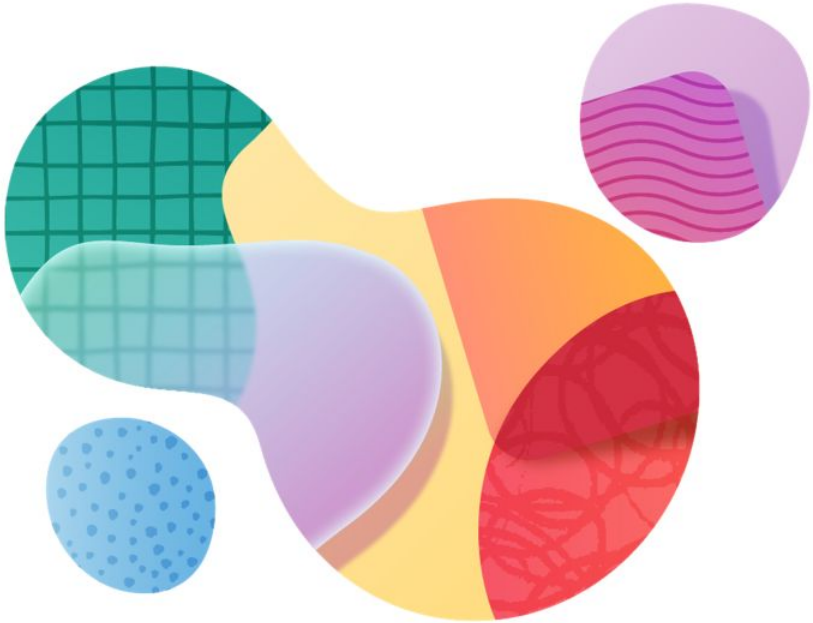
- ❑ The Multiple-Returns pattern often leads to shorter code.

- ❑ The Multiple-Returns pattern often leads to faster code.

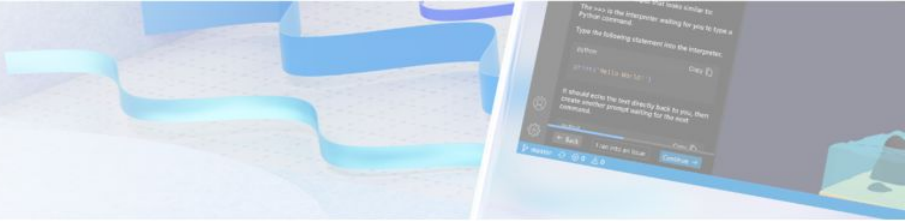
- ✓ The Build-Up-Return pattern only has one `return` statement, so the control flow is less complicated.
  Having multiple `return` statements makes the control flow more confusing because the function could end at any point — you must keep track of which lines were skipped.

Define-and-Refine Pattern

- The `else` statement is a powerful way to specify default behavior after `if` and `elif` statements.
- When you nest these statements, the `else` branch isn't always available.
- The nested statements represent specific, narrow cases involving multiple conditions.



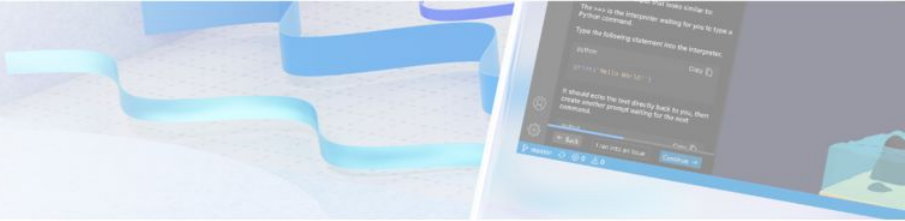
Define-and-Refine Pattern



```
if ___:
    if ___:
        value = __special__
    else:
        value = __default__
else:
    value = __default__
# Becomes
value = __default__
if ___:
    if ___:
        value = __special__
```

You can use the Define-and-Refine pattern to specify an initial value for a variable, and then only specify the branches to change that value.

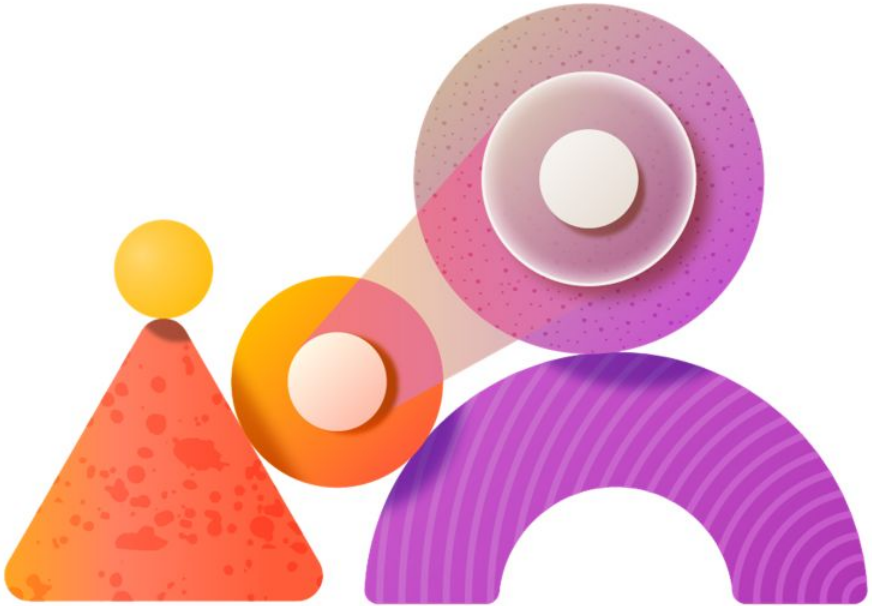
Define-and-Refine Pattern



- This code is easier to read because it doesn't have an `else` statement for every `if` statement.
- By initializing a default value for `weather`, you avoid having any `else` statements.

```
weather = 'unpleasant'
if temperature > 50:
    if temperature < 80:
        if rainfall == 0:
            weather = 'nice'
# Or equivalently
weather = 'unpleasant'
if temperature > 50 and temperature < 80 and rainfall == 0:
    weather = 'nice'
```

Define-and-Refine Pattern



- There's value in having an explicit `else` case so that future developers understand what you meant.
- Consider whether the pattern improves the code or makes the code as readable as possible.

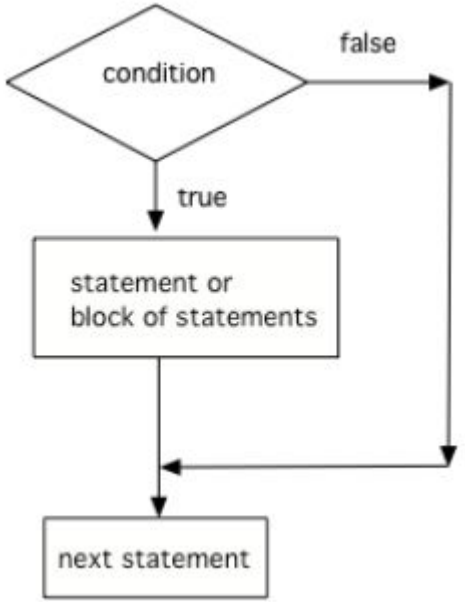
For Loops

For Loops

We introduced the concept of controlling the "flow" of your program by introducing branches via **conditionals.**

Another concept that is fundamental to **processing data** is *iteration* or *looping*.

Programs cannot interact with more than one piece of data at a time. Therefore, we must introduce syntax which can **iterate** over a sequence of data.



Overview

This program prints a list of numbers with each number on its own line, in two ways:

1. Write a `print` statement five times, once for each number.

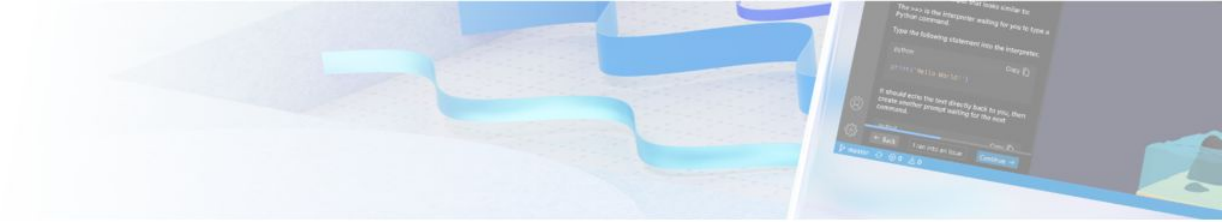
2. Write a pair of statements by using a `for` loop.

```
costs = [100, 25, 25, 50, 10]

# Without FOR loops
print(costs[0])
print(costs[1])
print(costs[2])
print(costs[3])
print(costs[4])

# With FOR loops!
for a_cost in costs:
    print(a_cost)
```

1

2

Overview



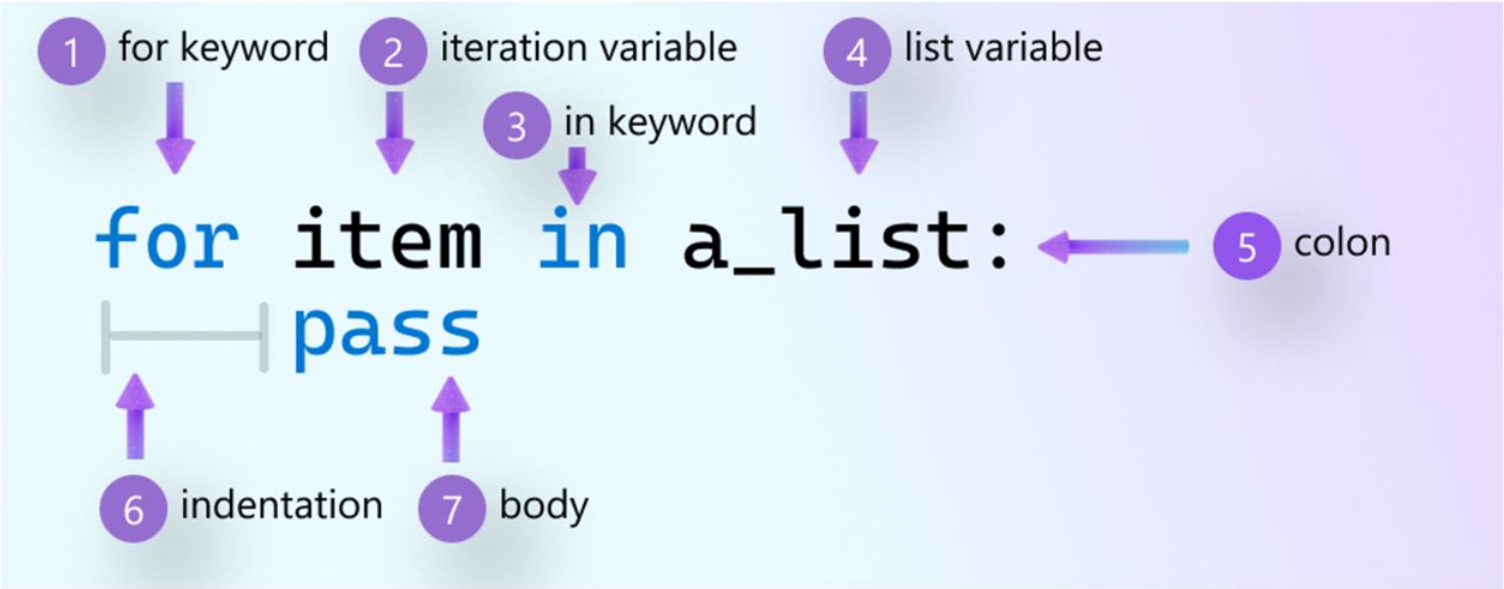
Using the example, here's how you write a `for` loop:

1. Write the word `for`.
2. Make a new variable, named the *iteration variable*.
3. Write the word `in`.
4. Put the list variable or value that you want to iterate over.
5. Put a colon (`:`).
6. Put statements inside the body of the loop, indented with 4 spaces.

```
costs = [100, 25, 25, 50, 10]

# Without FOR loops
print(costs[0])
print(costs[1])
print(costs[2])
print(costs[3])
print(costs[4])

# With FOR loops!
for a_cost in costs:
    print(a_cost)
```

Overview

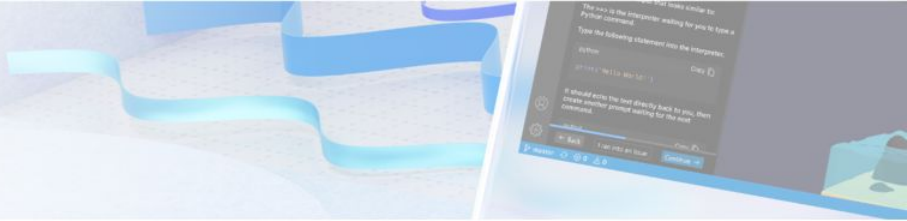
The first line of the `for` loop (the parts before the body) is called the *loop header*.



Iteration List

- A crucial element of any `for` loop is the *iteration list* — the data that you want to iterate over.
- When the loop executes, Python assigns each list item to the iteration variable in turn, whether there are no items or many.

Check Your Understanding

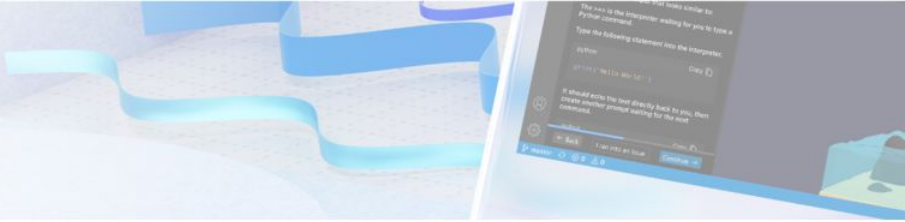


Question 1

How many times will a `for` loop iterate when given an empty list?

- ❑ One time
- ❑ There's no way to know
- ❑ Zero times

Check Your Understanding



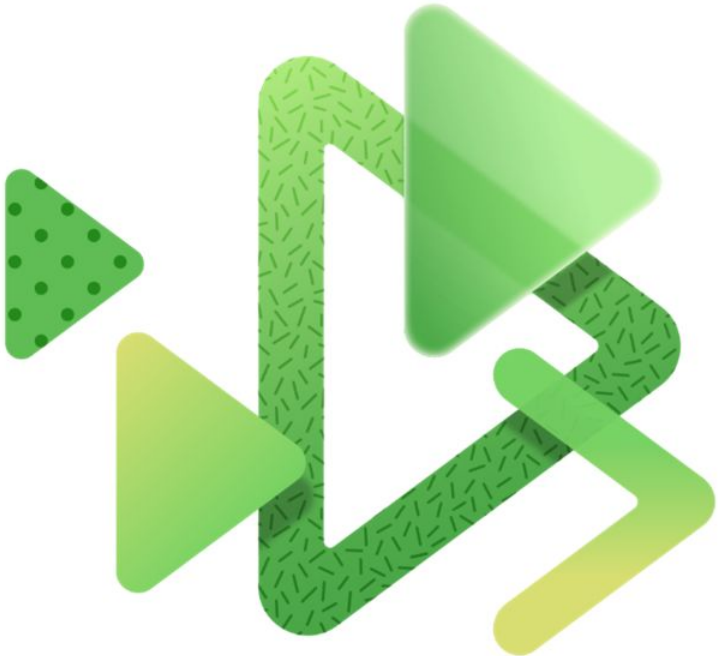
Question 1

How many times will a `for` loop iterate when given an empty list?

- ❑ One time
- ❑ There's no way to know
- ✓ Zero times

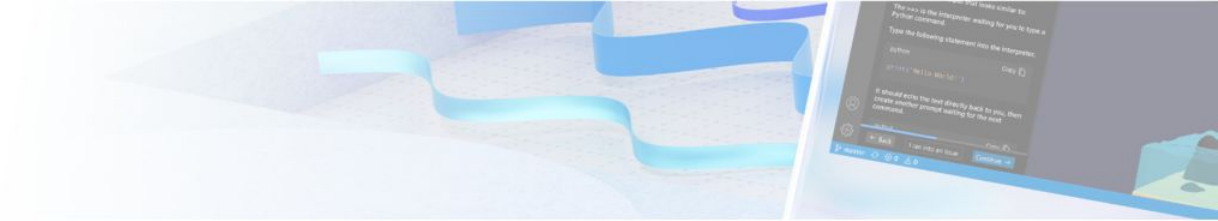
  Because a `for` loop iterates once for each list element, a list with zero elements causes it to loop zero times.

Iteration Variable



- The *iteration variable* represents the generalized version of each list item in the list.
- For example, if you have a list of temperatures, it's "*a* temperature."
- By performing operations on this generalized version of a list item, you can concentrate on one element at a time.

Iteration Variable



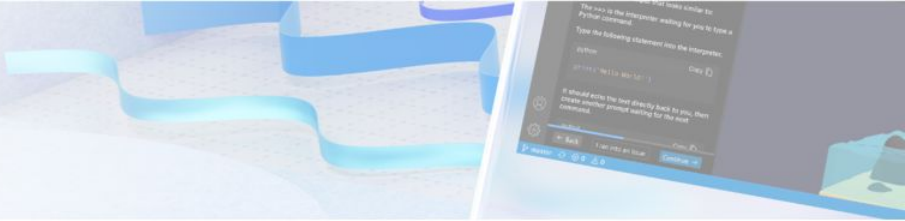
- Sometimes referred to as the *iteration target*.

- Two critical facts to remember:
  - The `for` loop creates the iteration variable when it executes.
  - The type of the iteration variable is the same as the element type of the list.

- The example shows examples of `for` loops with well-named iteration variables.

```
for a_temperature in temperatures:
    pass

for a_book in books:
    pass

for a_name in names:
    pass

for a_fruit in fruits:
    pass
```

Check Your Understanding



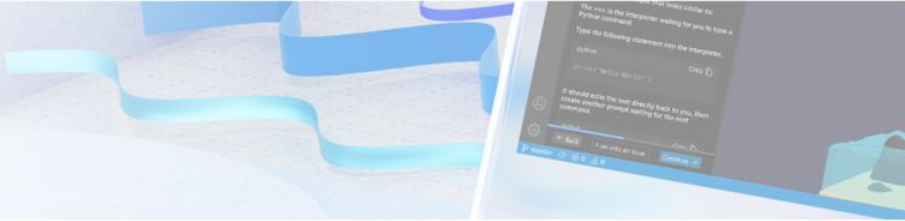
Question 2

What is the type of the iteration variable in this code?

```
digits = ["9", "3", "6"]
for digit in digits:
  print(digit)
```

- ❑ `str`
- ❑ `int`
- ❑ `list[str]`

Check Your Understanding



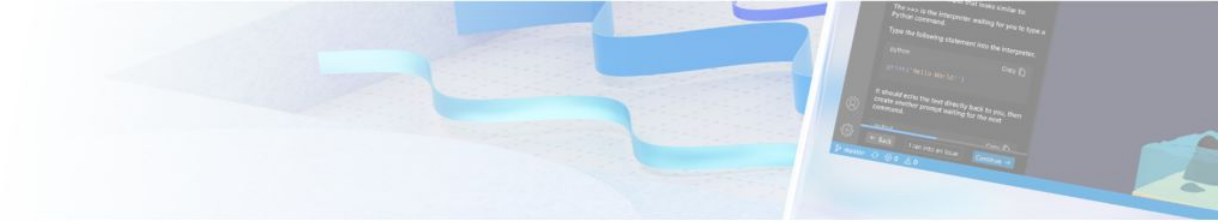
Question 2

What is the type of the iteration variable in this code?

```
digits = ["9", "3", "6"]
for digit in digits:
  print(digit)
```

- ✓ `Str`
  Because the element type of the list is `str`, the iteration variable is also a `str`.
- ❑ `int`
- ❑ `list[str]`

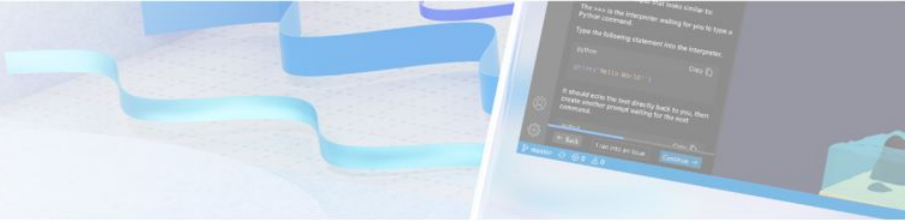
The Body



```
prices = [10, 20, 15]

for a_price in prices:
    adjusted = a_price * .9
    print(adjusted)
# Body ends
```

- Like `if` statements, you can put statements inside a `for` loop.
- Each statement is indented with 4 spaces and is executed one by one, from top to bottom. The first unindented line marks the end of the body.
- Statements are repeated once for each element of the list.

Check Your Understanding



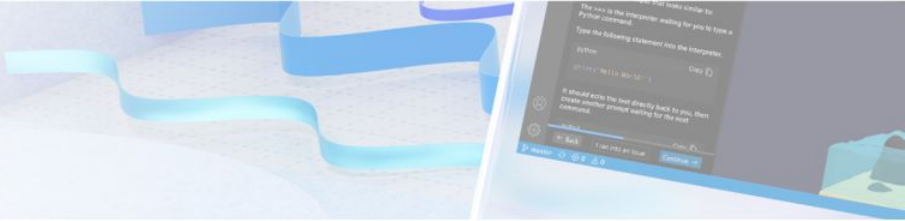
Question 3

Given the code shown here that iterates over the prices list, how many statements are executed in total inside of the loop body?

```
prices = [10, 20, 15]

for a_price in prices:
    adjusted = a_price * .9
    print(adjusted)
# Body ends
```

- ❑ 2 statements
- ❑ 3 statements
- ❑ 6 statements

Check Your Understanding



Question 3

Given the code shown here that iterates over the prices list, how many statements are executed in total inside of the loop body?

```
prices = [10, 20, 15]

for a_price in prices:
    adjusted = a_price * .9
    print(adjusted)
# Body ends
```

- ❑ 2 statements
- ❑ 3 statements
- ✓ 6 statements

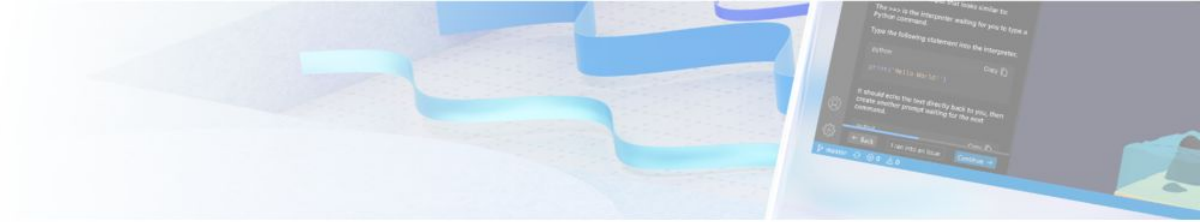
  Each of the 3 list elements in the list has 2 statements executed, resulting in 6 total statements executed.

Flow of a `for` Loop

- Like `if` statements, `for` loops also make the stream split into two directions, but one of the new streams moves back up — the reason why it's called a loop.
- The program *loops* until each list element is assigned to the iteration variable.

```
for a_price in prices:
    adjusted = a_price * .9
    print (adjusted)
```

Scope of a `for` Loop



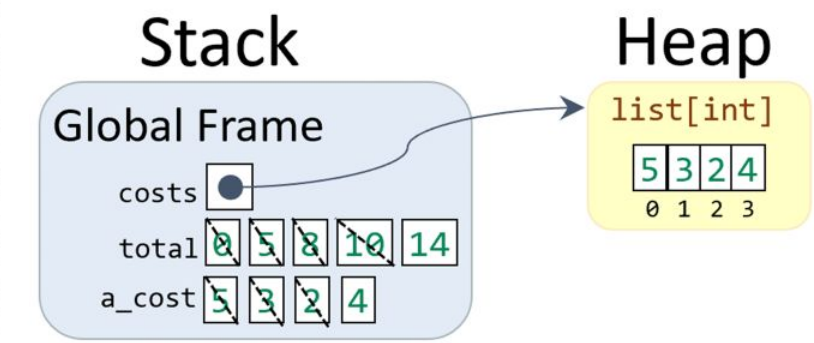
```
numbers = [1, 5, 2]
message = "The number is"
for number in numbers:
    print(message, number)
print("The last number was", number)
```

- Unlike functions, `for` loops don't have their own scope: Variables defined outside a `for` loop can be used inside the loop body.
- You can use the iteration variable after the loop finishes.

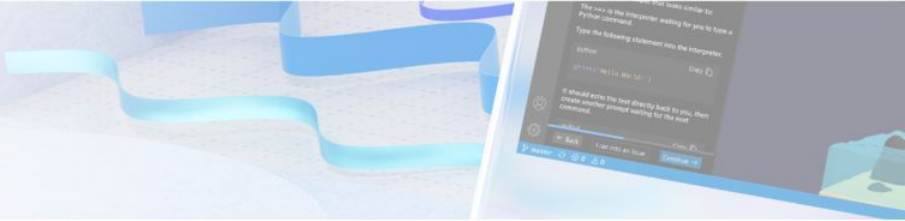
Tracing a `for` Loop

This stack/heap diagram shows an iteration over a list, adding each number to a running sum that's finally printed at the end.

```
costs = [5, 3, 2, 4]
total = 0
for a_cost in costs:
    total += a_cost
print(total)
```



Check Your Understanding



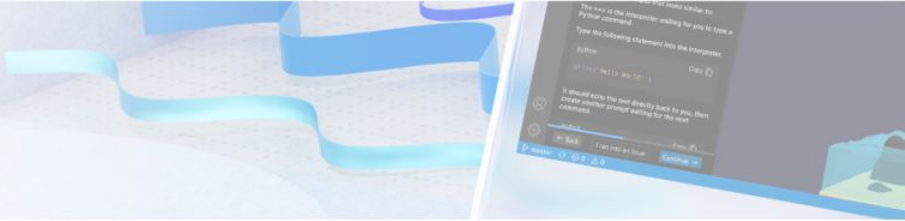
Question 3

Given the code shown here, what are the final values that are stored in `grade`, `adjusted`, and `total`?

```
grades = [10, 20]
total = 0
for grade in grades:
    adjusted = grade + 10
    total = total + adjusted
```

- ❑ `total` is 50, `adjusted` is 30, and `grade` is 20.
- ❑ `total` is 30, `adjusted` is 20, and `grade` is10.
- ❑ The `grade` and `adjusted` variables won't exist after the loop finishes because they weren't defined in the scope of the `for` loop.

Check Your Understanding



Question 3

Given the code shown here, what is the final value that's stored in `grade`, `adjusted`, and `total`?

```
grades = [10, 20]
total = 0
for grade in grades:
    adjusted = grade + 10
    total = total + adjusted
```

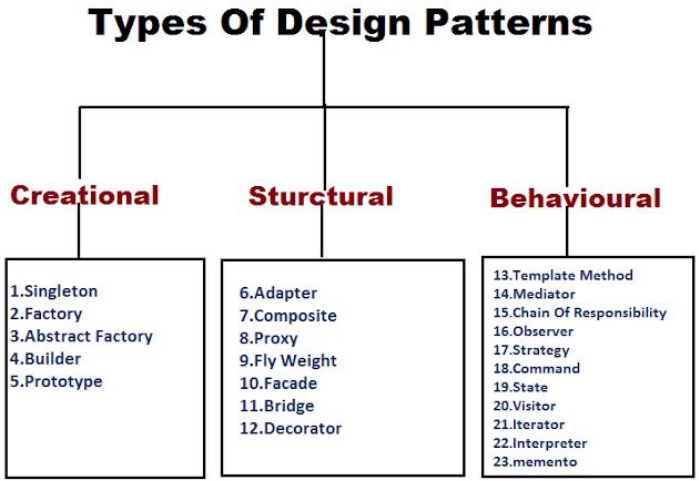
- ✓ `total` is 50, `adjusted` is 30, and `grade` is 20.
- ❑ `total` is 30, `adjusted` is 20, and `grade` is10.
- ❑ The `grade` and `adjusted` variables won't exist after the loop finishes because they weren't defined in the scope of the `for` loop.

For Loop Patterns

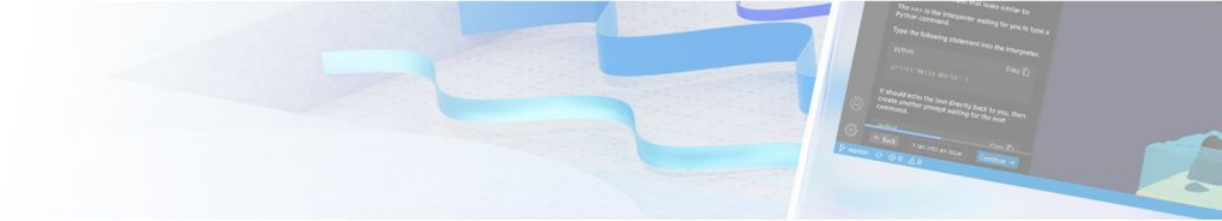
Coding Patterns

As we continue to learn more about various syntax, we will always follow it up with **patterns.**

There are many different ways to accomplish the same task in programming, however more often than not, there is usually **one specific** (and efficient) pattern that we should follow.



Overview

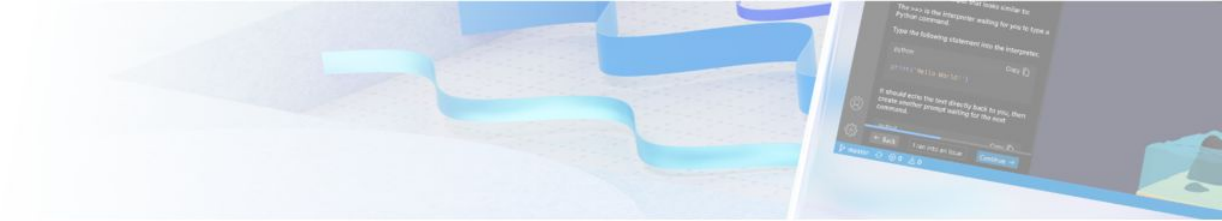


- This lesson introduces a few common patterns used with `for` loops.

- Think of these patterns as templates that you can adapt and combine to solve more complex problems.

  - **Count**: Count the number of elements in a list.
  - **Sum**: Add up a list of numbers.
  - **Accumulate**: Combine a list of strings or Booleans.
  - **Map**: Transform the elements of a list.

Count Pattern



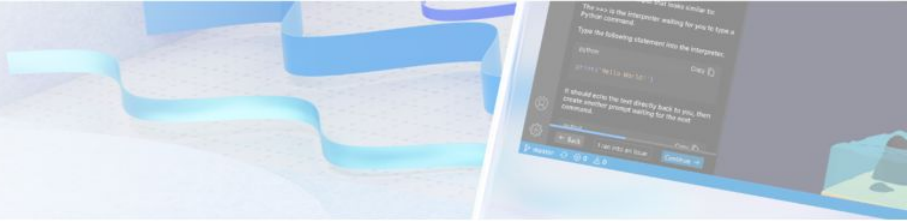
You want to know how many items are in your list. Here's a simple algorithm:

1. Start with an initial value of 0.
2. For each element, add 1 to a `count` variable.
3. When the loop finishes, the `count` variable has the length of the list.

```
a_list = ["Alpha", "Beta", "Gamma"]

count = 0
for item in a_list:
    count = count + 1

print(count)
```

Note that the item iteration variable is never actually used.

Check Your Understanding



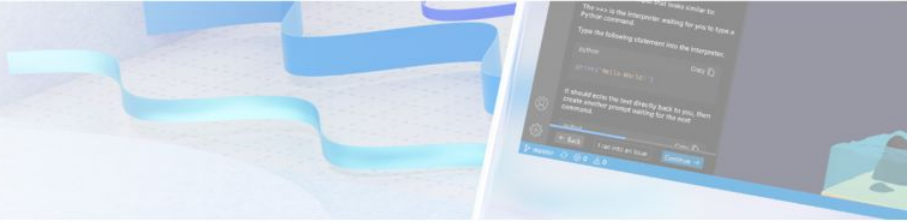
Question 1

After the third iteration of the loop, what is the value of count?

- ❑ 0
- ❑ 3
- ❑ 6
- ❑ 12

```
a_list = ["Alpha", "Beta", "Gamma"]


count = 0
for item in a_list:
  count = count + 1


print(count)
```

Check Your Understanding



Question 1

After the third iteration of the loop, what is the value of count?

- ❑ 0
- ✓ 3
  The statement is indented inside of the loop body.
- ❑ 6
- ❑ 12

```
a_list = ["Alpha", "Beta", "Gamma"]


count = 0
for item in a_list:
  count = count + 1


print(count)
```

Sum Pattern

- You want to add up all of the numbers in your list.
- The plus operator can take only two items at a time, so you add each element one at a time to a `sum` variable, initialized to 0.
- The `sum` pattern is like the `count` pattern, except that instead of adding 1, you add the iteration variable.

```
a_list = [10, 30, 20]

sum = 0
for item in a_list:
    sum = sum + item

print(sum)
```

Check Your Understanding

Question 2

After the third iteration of the loop, what is the value of sum?

- ❑ 0
- ❑ 3
- ❑ 6
- ❑ 12

```
sum = 0
values = [4, 2, 6]
for value in values:
    sum = sum + value
```

Check Your Understanding

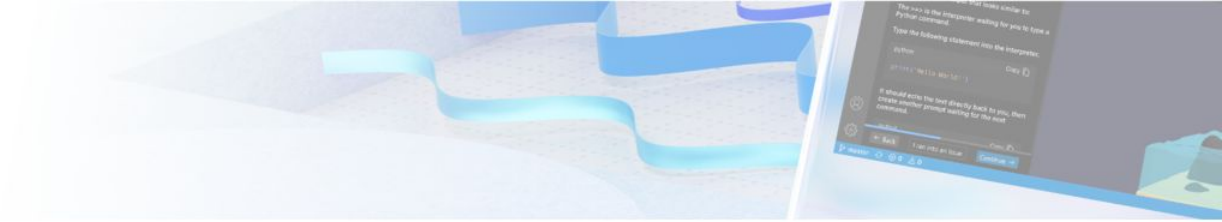
Question 2

After the third iteration of the loop, what is the value of sum?

```
sum = 0
values = [4, 2, 6]
for value in values:
    sum = sum + value
```

- ❑ 0
- ❑ 3
- ❑ 6
- ✓ 12

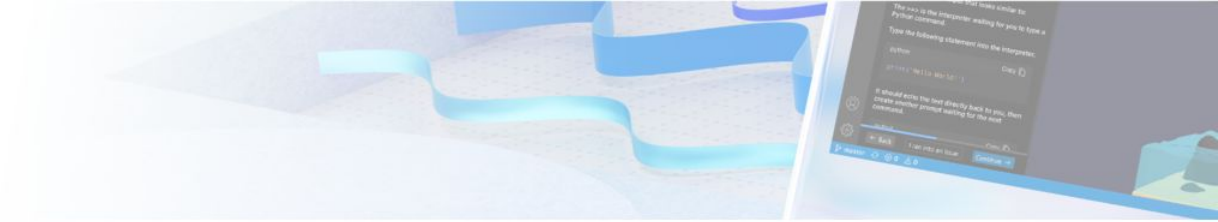
Accumulate Pattern



```
result = ___
for item in a_list:
    result = result ___ item
```

- The `sum` and `count` patterns are both more specific examples of the `accumulate` pattern.
- You can use the `accumulate` pattern to start with an initial value and use any function or operation that takes in two values.
- This pattern can be applied to numbers, strings, Booleans, and any type that can be combined with an operator that takes two operands.

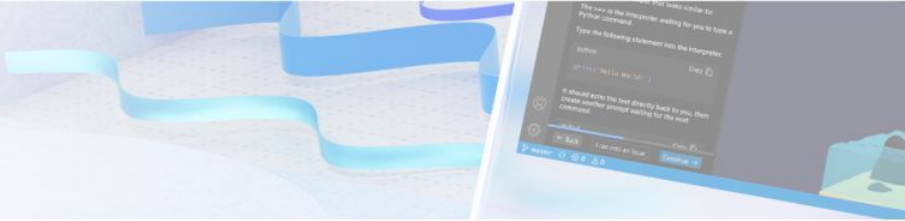
String Accumulation



- *String accumulation* is the process of gradually building up a string by adding more characters or other strings to it.
- The only difference between string accumulation and the summation pattern is that it adds strings together instead of integers.
- The result is a single string with all source strings joined together.

```
sentence = ""  # Start with an empty string
words = ["Hello, ", "how ", "are ", "you?"]

# Add the words one by one
for word in words:
    sentence += word

print(sentence)
# Output: Hello, how are you?
```

Check Your Understanding



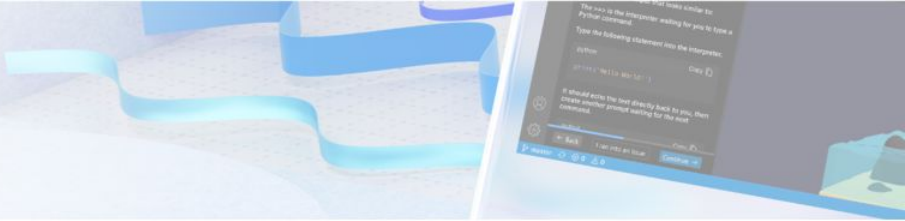
Question 3

What does the code here print?

- ❑ "Carol"
- ❑ "Alice, Bob, Carol,"
- ❑ "Alice, Bob, Carol"

```
names = ["Alice", "Bob", "Carol"]
added_names = ""
for name in names:
    added_names = added_names + name + ", "
print(added_names)
```

Check Your Understanding



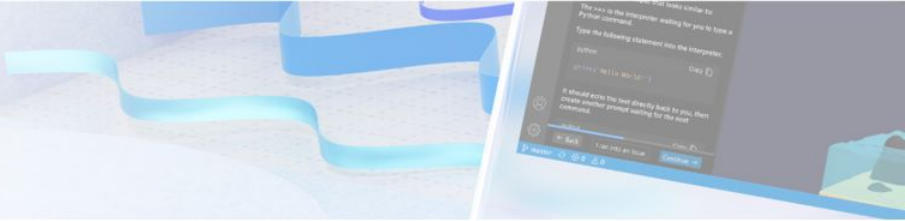
Question 3

What does the code here print?

```
names = ["Alice", "Bob", "Carol"]
added_names = ""
for name in names:
    added_names = added_names + name + ", "
print(added_names)
```

- ❑ "Carol"
- ✓ "Alice, Bob, Carol"
  If you're not careful with how you combine strings, you'll end up with an extra comma.
- ❑ "Alice, Bob, Carol,"

Boolean Accumulation (any/all)

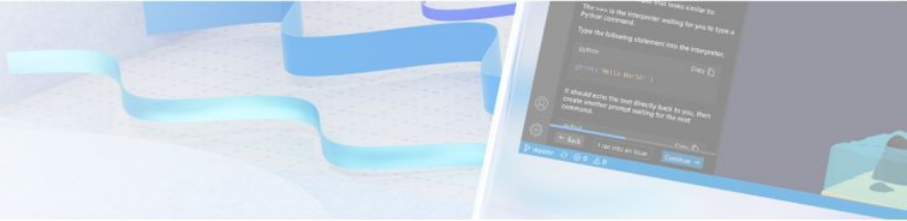


Any variant

- Boolean accumulation patterns come in two versions: *any* and *all*.
- *Any* starts with an initial value of False, and then you check whether either the current element or the accumulation variable is True.
- If you encounter a True value in the list, the accumulation variable becomes True and evaluates to True in every subsequent check.

```
answers = [True, True, False, True]

# Any True?
any_true = False
for answer in answers:
    any_true = any_true or answer
print(any_true)
```

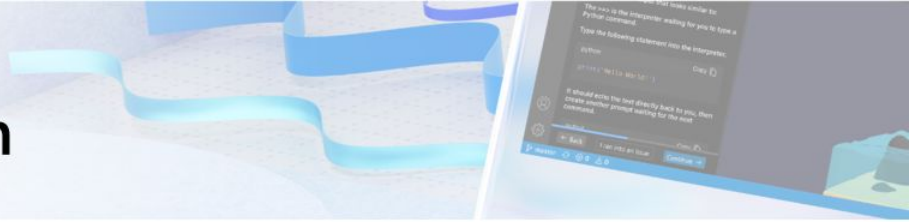
Boolean Accumulation (any/all)



The *all* variant starts with the initial value of True. If you encounter a False value in the list, the accumulation variable becomes False and evaluates to False in every subsequent check.

```
answers = [True, True, False, True]


# All True?
all_true = True
for answer in answers:
    all_true = all_true and answer
print(all_true)
```

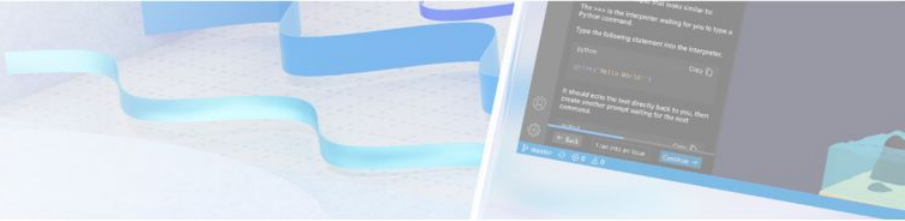
Boolean Accumulation in Action



- The Boolean accumulation pattern works on more than just a list of Boolean values.
- You can use an expression that produces a Boolean to ask a question about each element of a list.
- No `if` statement is required — you simply use `or` to combine logical conditions and return the final Boolean result.

```
from bakery import assert_equal

def any_red(colors: list[str]) -> bool:
    result = False
    for color in colors:
        result = result or color == "red"
    return result

assert_equal(any_red(["blue", "red", "green"]), True)
assert_equal(any_red(["r", "g", "b"]), False)
```

Check Your Understanding



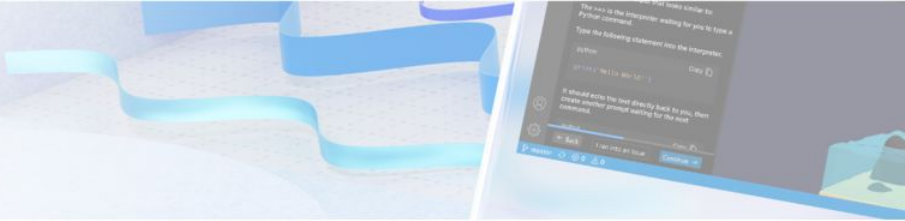
Question 4

What is the value of `larges` after the following Boolean accumulation is executed?

```
sizes = ["M", "S", "L", "XL"]
larges = False
for size in sizes:
  larges = larges or size == "L"
print(larges)
```

- ❑ [False, False, True, False]
- ❑ True
- ❑ False

Check Your Understanding



Question 4

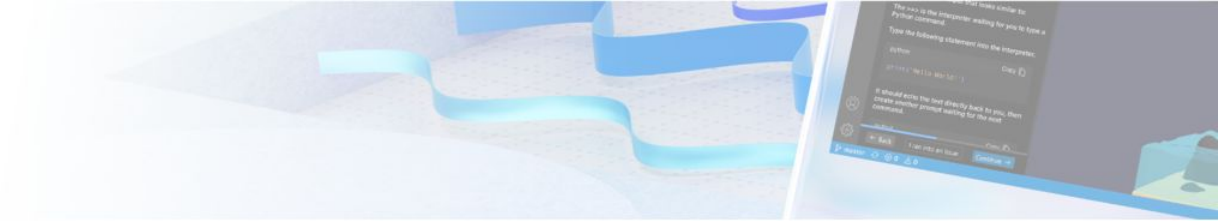
What is the value of `larges` after the following Boolean accumulation is executed?

```
sizes = ["M", "S", "L", "XL"]
larges = False
for size in sizes:
  larges = larges or size == "L"
print(larges)
```

- ❑ [False, False, True, False]
- ✓ True

  Because even just one element of the list was equal to *L*, the value stored in `larges` becomes True and stays that way.
- ❑ False

Map Pattern

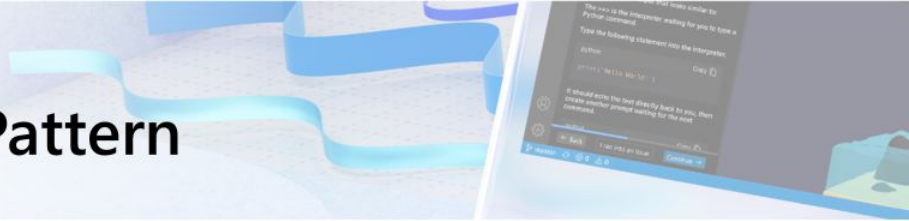


- When you accumulate a list and start with an empty list as your initial value, and then append each value one at a time, you end up with a copy of the original list.

- You can further modify the new list without worrying about changing the original list. This is the key idea of the map pattern.

```
old_list = ["Apple", "Grape", "Orange"]

copied_list = []
for item in old_list:
    copied_list.append(item)

print(copied_list)
```

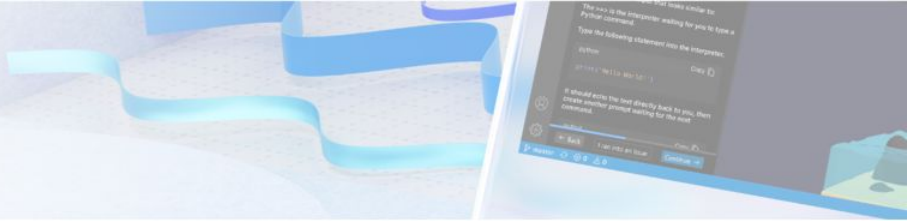
Modifying a List with the Map Pattern



- As you append values, you can also modify them.
- This program simply adds the string `s` to the end of each fruit type to make the plural version of the word.
- You can transform each element of a list by using any valid operation or expression as arguments to the `append` method.

```
fruits = ["Apple", "Grape", "Orange"]

plural_fruits = []
for fruit in fruits:
    plural_fruits.append(fruit + "s")

print(plural_fruits)
```

Accidental Infinite Loop



- If you try running the program shown here, your program will take a very long time — maybe forever — because you're appending to the same list that you're iterating over.

- If you append to the fruits list while you iterate through the list, the `for` loop keeps finding new elements to process until you stop the program.

```
fruits = ["Apple", "Grape", "Orange"]

plural_fruits = []
for fruit in fruits:
    plural_fruits.append(fruit + "s")

print(plural_fruits)
```

Inside or Outside of the Body

- Remember, every statement inside the body is executed for each element. Put things inside only if they should happen for each element.

- The patterns can help you keep track of where things go, but ultimately, you must think critically to know for sure.

```
# Before
for item in a_list:
    # Inside
    pass
# After
```

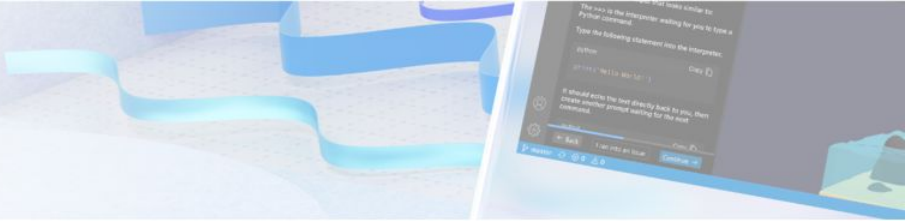
Inside or Outside of the Body

- For patterns inside of functions that consume lists, all of the same rules about indentation and scope apply, but it might not be clear where the `return` statement goes relative to the body of the `for` loop.

- In the example, notice that the `return` statement goes *after* the body of the for loop, *outside* of the `if` statement but *inside* the body of the function definition.

```
from bakery import assert_equal

def add_3(numbers: list[int]) -> int:
    result = []
    for number in numbers:
        result.append(number + 3)
    return result

assert_equal(add_3([1, 2, 3]), [4, 5, 6])
assert_equal(add_3([3, 3, 3, 3]), [6, 6, 6, 6])
assert_equal(add_3([]), [])
```

Check Your Understanding



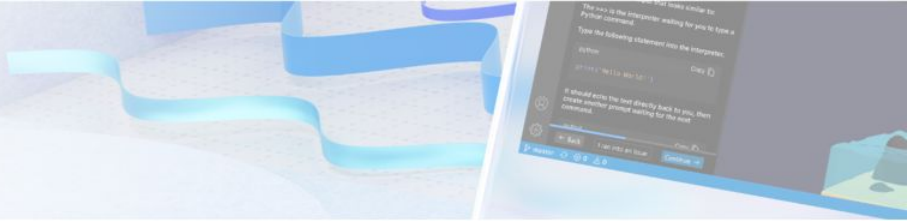
Question 5

Is the `print` call before, inside, outside, or after the following for loop?

```
for name in names:
    print(name)
```

- ❑ Before
- ❑ After
- ❑ Inside
- ❑ Outside

Check Your Understanding



Question 5

Is the `print` call before, inside, outside, or after the following for loop?

```
for name in names:
    print(name)
```

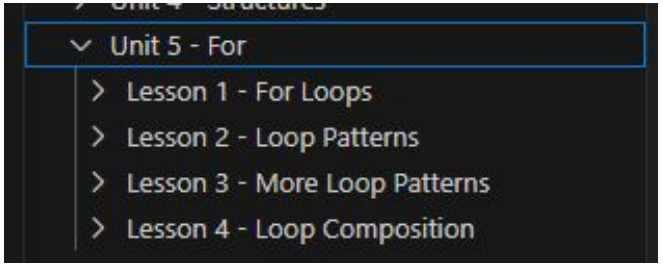
- ❑ Before
- ❑ After
- ✓ Inside
  The statement is indented inside of the loop body.
- ❑ Outside

In-Class Lab

Lab - GitHub Module

For the remaining lab time, break into your pod groups and complete the following modules:

- **VSCode Unit 5: Lesson 1 - Lesson 3**



Wrap-Up

Lab (Due 03/28)

Taipei City, Taiwan

The company you work for, Seng-Links, aims to identify periods when a user sleeps or exercises using their varying recorded heart rates.

Your company has provided you a data folder (*data/*) of **5 files** that contain heart-rate samples from a participant. The participants device records heart rate data every 5 minutes (aka *sampling rate*).

You are tasked with writing code that **processes each data file**. You will utilize test-driven development in order to complete this project.

Wednesday...

On Wednesday we will go over:...

- How to ingest text data via Python.
- How to navigate your computer via both relative and full paths.
- Additional for loop patterns.

Jupyter: scratchpad of the data scientist

If you understand what you're doing, you're not learning anything. - Anonymous

Glossary