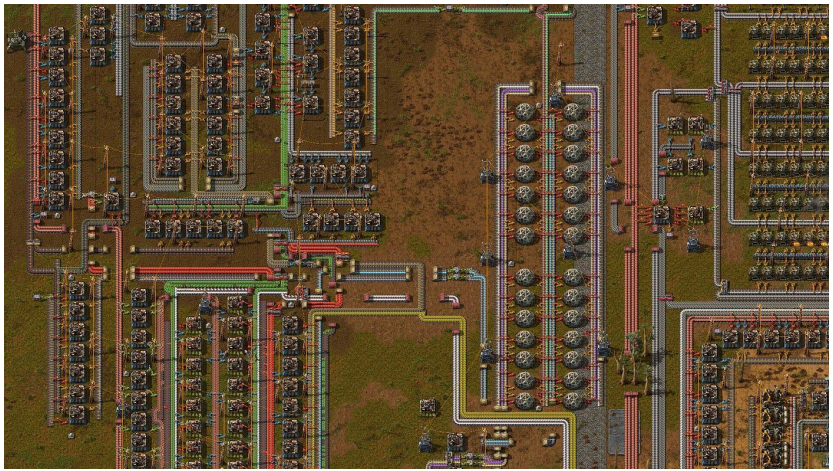
Introduction to Data Processing

Agenda - Schedule

1. Warm-Up
2. Advanced For Loop Patterns
3. File I/O
4. Lab



Data engineering is the art of ingesting data without error.

Agenda - Goals

- Implement advanced for-loop patterns in Python
- Understand how to interpret paths on your operating system
- Understand how to work with text files in your Python program

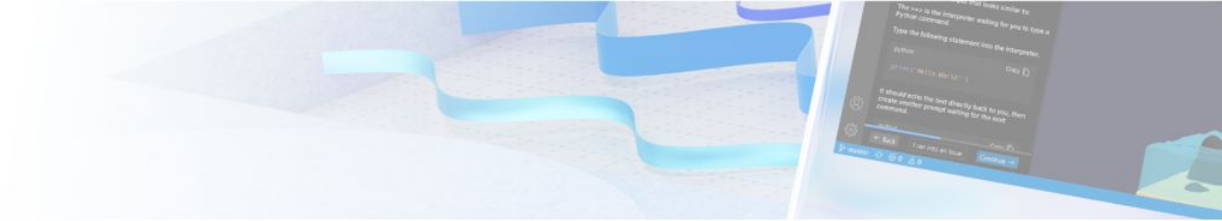
Warm-Up

```
def evaluate(in_list: list) -> list:
        new_list = []
        for val in in_list:
                if val > 100.4:
                        new_list.append("fever")
                elif val > 99.1 and val <= 100.4:
                        new_list.append("upper limit")
                else:
                        new_list.append("normal")
         return new_list


print(evaluate([100.3, 100.2, 99.9, 100.6, 100.7, 100.1]))
print(evaluate([99.0, 99.8, 98.1, 97,6, 98.5]))
```

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

Advanced For Loop Patterns

Overview

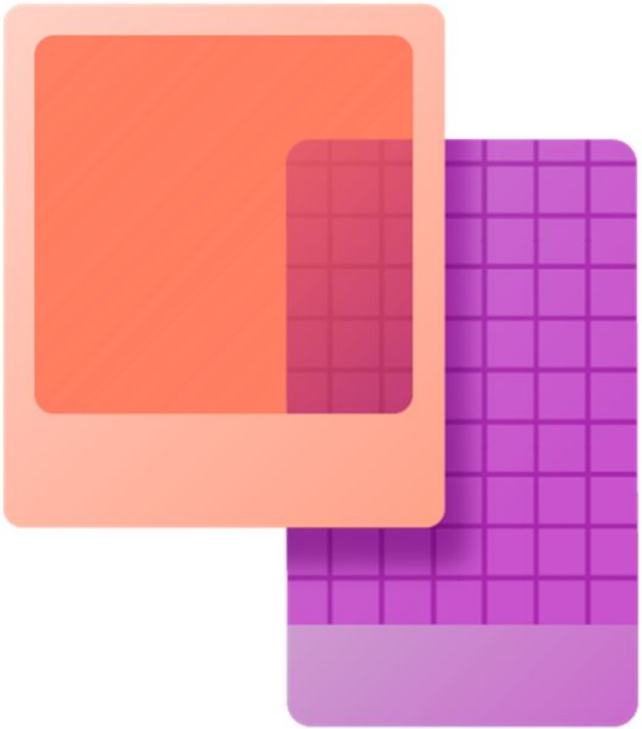


You've seen the **sum**, **count**, **accumulate**, and **map** patterns. You'll now learn about a few more patterns that incorporate conditionals:

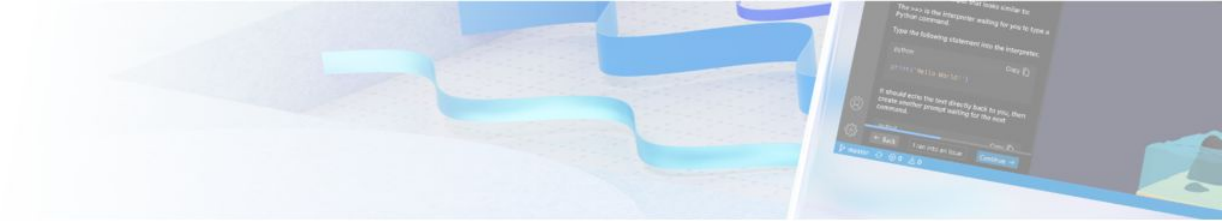
- **Filter**: Remove or keep elements that match a condition.
- **Find**: Get the first or last element that matches a condition.
- **Take**: Keep all elements until they match a condition.
- **Min/Max**: Find the highest or lowest element in a list.

Filter Pattern

- You have a list of numbers and want to ignore some of them according to a rule.
- By embedding an `if` statement inside the loop, you can optionally include or not include elements in your accumulation.
- This is the **filter** pattern, which is more of an optional modification you can make to other patterns.



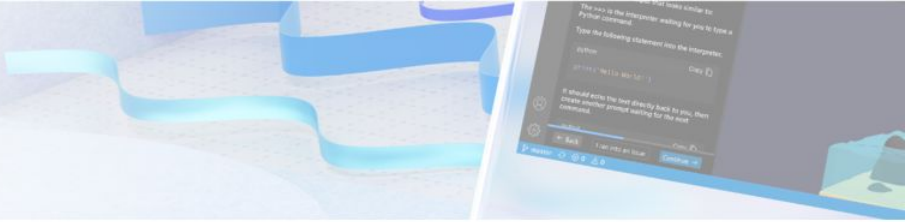
Filter Pattern



- The example shows the filter pattern combined with the map pattern.
- The body of the `for` loop repeats however many times there are items in the list.
- Each time, if the `if` statement's condition is True, Python executes its body with the `append` statement.
- When the `for` loop finishes, you execute the `print` statement after the `for` loop.

```
Numbers = [50, 100, 200]


# Filter+Map
new_list = []
for number in numbers:
    if number < 100:
        new_list.append(number)
print(new_list)
```

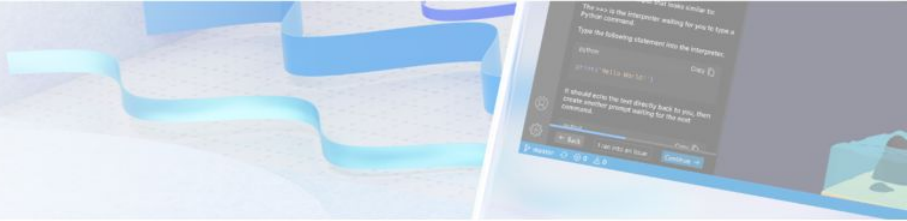
Filter Pattern with Count



- The filter pattern is very compatible with the other patterns.
- The example shows a variant of combining the filter pattern with the count pattern.
- The basic code structure is the same, with an `if` statement inside of the `for` loop.
- The big difference comes in the initial value and accumulation step, which match the count pattern instead of the map pattern.

```
Numbers = [50, 100, 200]


# Filter + Count
result = 0
for number in numbers:
    if number <= 100:
        result = result + 1
print(result)
```

Check Your Understanding



Question 1

In the loop in the example, how many times will Python check the `if` statement?

- ❑ 3
- ❑ 2
- ❑ 1

```
Numbers = [50, 100, 200]


# Filter + Count
result = 0
for number in numbers:
    if number <= 100:
        result = result + 1
print(result)
```

Check Your Understanding

Question 1

In the loop in the example, how many times will Python check the `if` statement?

- ❑ 1
- ❑ 2
- ✓ 3

  Python checks the `if` statement on every iteration of the loop. However, the `if` statement will only be True in two of those iterations.

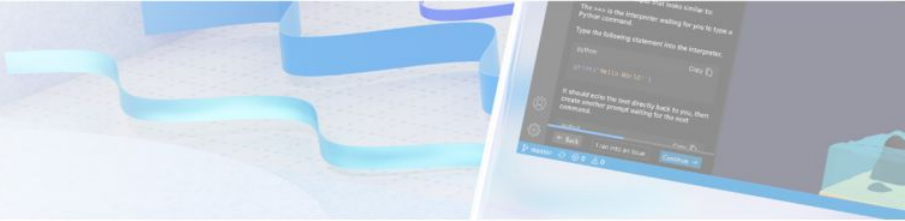
```
Numbers = [50, 100, 200]


# Filter + Count
result = 0
for number in numbers:
    if number <= 100:
        result = result + 1
print(result)
```

Find Pattern

- The example shows a list of items. You want to find a specific value based on a condition. This requires that you walk through the entire list and check whether that value satisfies your condition.
- When you're done, either the desired value is in the `found` variable, or `found` has the value `None`.
- You need to replace the blank with an appropriate condition, often `None` with a suitable default value.

```
found = None
for item in my_list:
    if ___:
        found = item
```

Check Your Understanding

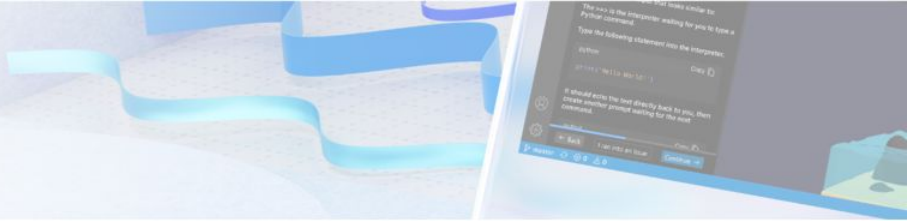


Question 2

What is the difference between the filter pattern and the find pattern?

- ❑ The filter pattern only works on numeric values, while the find pattern works on any type.
- ❑ The filter pattern returns a list, while the find pattern returns a single element from the list.
- ❑ One pattern uses a conditional and the other doesn't.

Check Your Understanding

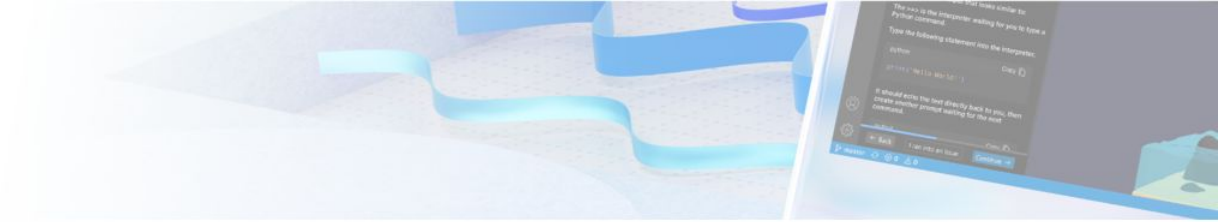


Question 2

What is the difference between the filter pattern and the find pattern?

- ❑ The filter pattern only works on numeric values, while the find pattern works on any type.
- ✓ The filter pattern returns a list, while the find pattern returns a single element from the list.
  Both patterns involve a conditional, but they return very different types of values.
- ❑ One pattern uses a conditional and the other doesn't.

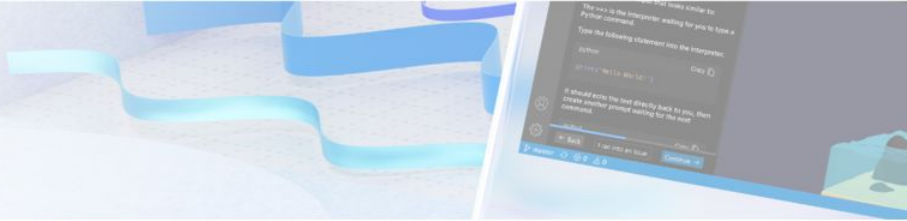
Find Last Variant



- The first example shows two variants of the find pattern.
- The first version finds the last element in the list that matches the given condition.
- The key idea is that loop repeatedly manipulates the accumulation variable and is only returned when the entire loop finishes.

```
from bakery import assert_equal

def find_last_question(words: list[str]) -> str:
    result = "Missing"
    for word in words:
        if word[-1] == "?":
            result = word
    return result

some_words = ["Hi?", "Oh.", "Uh?"]
assert_equal(find_last_question(some_words), "Uh?")
```

Check Your Understanding



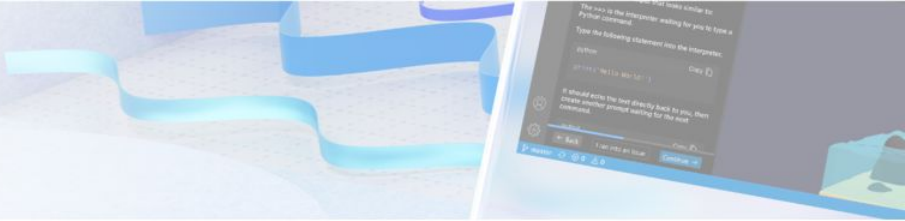
Question 3

After the following code is executed, what value is stored in `my_value`?

- ❑ "mY"
- ❑ "!"
- ❑ "MY"

```
my_value = None
values = ["MY", "my", "mY", "!"]
for value in values:
    if value.upper() == "MY":
        my_value = value
```

Check Your Understanding



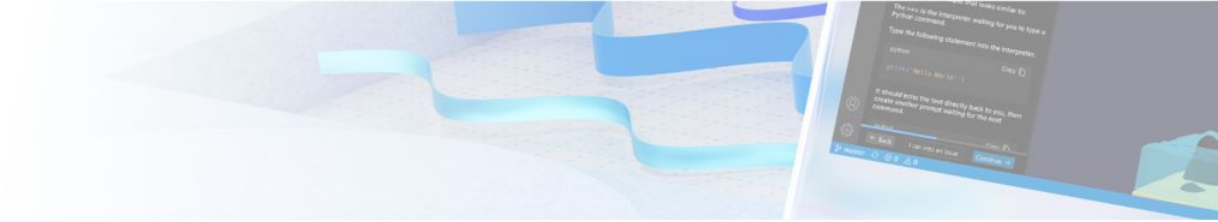
Question 3

After the following code is executed, what value is stored in `my_value`?

```
my_value = None
values = ["MY", "my", "mY", "!"]
for value in values:
    if value.upper() == "MY":
        my_value = value
```

- ✓ "mY"
  When the string gets converted to uppercase in the conditional, this value is equal to "MY". However, that doesn't change the string value stored in the value variable.
- ❑ "!"
- ❑ "MY"

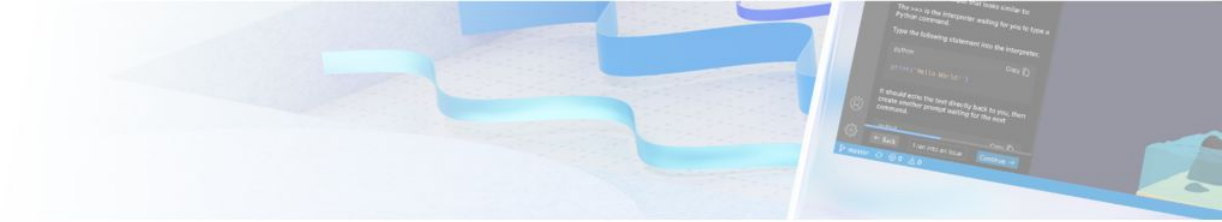
Find First Variant



- Contrast the previous example with the next variant of the find pattern: *find first*.
- In this variant, instead of returning the last element that matches the given condition, Python instead returns the first element it encounters.
- This variant relies on the control flow breaking behavior of the `return` statement to exit the `for` loop as soon as an element matches the condition.

```
from bakery import assert_equal

def find_first_question(words: list[str]) -> str:
    for word in words:
        if word[-1] == "?":
            return word
    return "Missing"

some_words = ["Hi?", "Oh.", "Uh?"]
assert_equal(find_first_question(some_words), "Hi?")
```

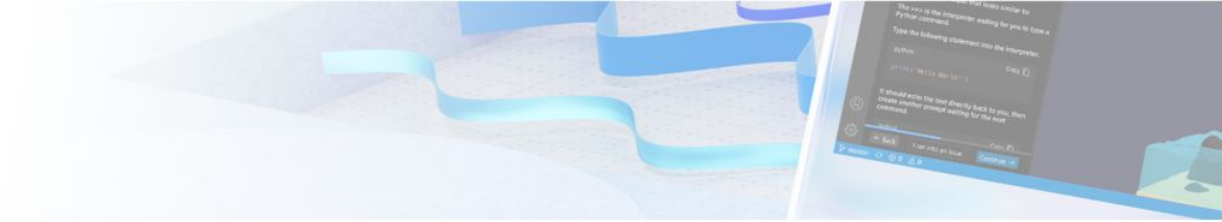
Find First Variant



- When encountered, the `return` statement immediately ends the `for` loop and the entire function.
- This is why it's critical to make sure that you always guard your `return` statements by using `if` statements.
- You'll rarely see a `return` statement directly inside of a `for` loop *unless* it's wrapped in an `if` statement.

```
from bakery import assert_equal

def find_first_question(words: list[str]) -> str:
    for word in words:
        if word[-1] == "?":
            return word
    return "Missing"

some_words = ["Hi?", "Oh.", "Uh?"]
assert_equal(find_first_question(some_words), "Hi?")
```

Take Pattern



Notice that the body of the loop is more complicated: It uses an `elif` statement to prevent you from taking elements as soon as you've met the condition.

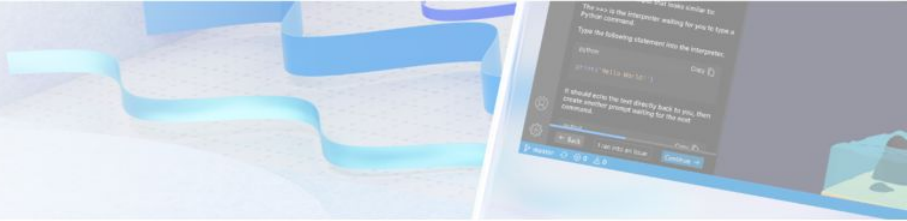
```
Taking = True
new_list = []
for item in my_list:
    if ___:
        taking = False
    elif taking:
        new_list.append(item)
```

The Take Pattern in Action

- After finding the first element that matched your condition, the new list stopped accumulating elements.
- At the end, the fact that Python took only the first two elements is a consequence of their data, not some arbitrary constant length.

```
words = ["Oh", "Hi", "There!", "Okay", "Bye!"]
taking = True
first_words = []
for word in words:
    if word[-1] == "!":
        taking = False
    elif taking:
        first_words.append(word)

print(first_words)
# ['Oh', 'Hi']
```

Check Your Understanding



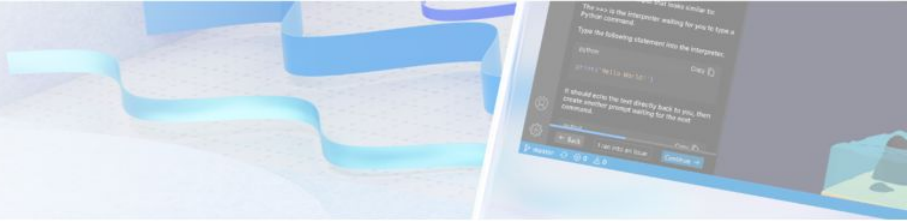
Question 4

What is printed after the example code is executed?

- ❑ [`Ada`, `Babbage`, `Captain? `]
- ❑ [`Ada`, `Babbage`]
- ❑ [`Ada`, `Babbage`, `Pico`]

```
names = ['Ada', 'Babbage', 'Captain?', 'Domino?', 'Pico']
known_names = []
taking = True
for name in names:
    if "?" in name:
        taking = False
    elif taking:
        known_names.append(name)
print(known_names)
```

Check Your Understanding



Question 4

What is printed after the example code is executed?

```
names = ['Ada', 'Babbage', 'Captain?', 'Domino?', 'Pico']
known_names = []
taking = True
for name in names:
    if "?" in name:
        taking = False
    elif taking:
        known_names.append(name)
print(known_names)
```

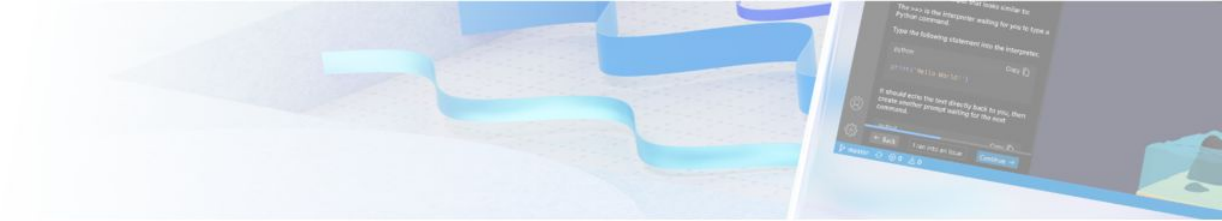
- ❑ [`Ada`, `Babbage`, `Captain? `]
- ✓ [`Ada`, `Babbage`]
  Only the first two elements are taken.
- ❑ [`Ada`, `Babbage`, `Pico`]

Min/Max Pattern

- You use the last pattern, min/max, to find the highest or lowest element in a list.
- Min/max is like the find pattern, except that the conditional compares each value to the accumulated value.

```
Maximum = a_list[0]
for item in a_list:
    if item > maximum:
        maximum = item

print(maximum)
```

Min/Max Pattern



- When you use this pattern, the accumulated value (in this case a maximum) ends up as the highest value in the list.
- The corresponding minimum pattern works similarly, except that it uses the less-than operator (<) instead of the greater-than (>) operator.

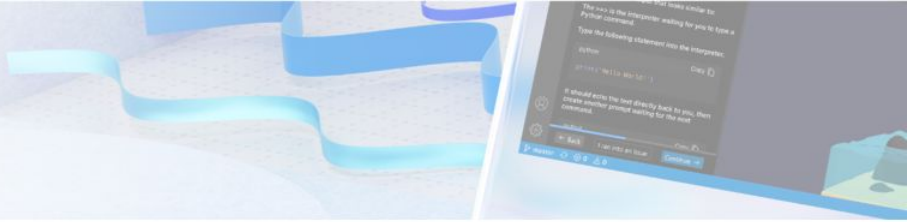
```
Maximum = a_list[0]
for item in a_list:
    if item > maximum:
        maximum = item

print(maximum)
```

Maximum of Non-Empty Lists

- Min/max patterns require that the list not be empty before you initialize the accumulation variable.
- Remember: You can't index the first element of an empty list because there's no first element in an empty list.
- Most practical implementations begin by incorporating the guard pattern you learned about in logical patterns.

```
def highest(numbers: list[int]) -> int:
    # Critical guard!
    if not numbers:
        return 0
    # Pattern continues from here
    maximum = numbers[0]
    for number in numbers:
        if number > maximum:
            maximum = number
    return maximum
```

Check Your Understanding

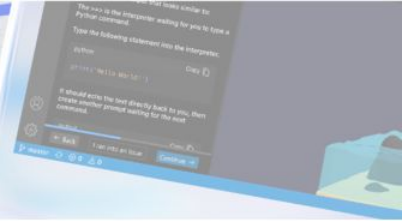


Question 5

Which of the patterns fails on an empty list?

- ❑ Filter
- ❑ Find
- ❑ Min/Max

Check Your Understanding



Question 5

Which of the patterns fails on an empty list?

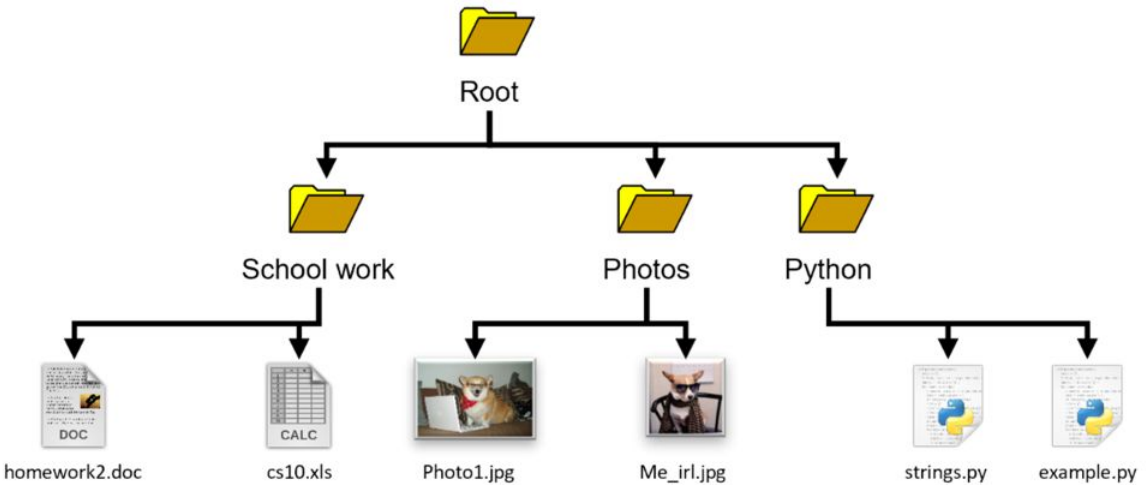
- ❑ Filter
- ❑ Find
- ✓ Min/max

  The min/max patterns expect the list to be non-empty. Otherwise, they can't initialize themselves to the first element of the list.

Paths

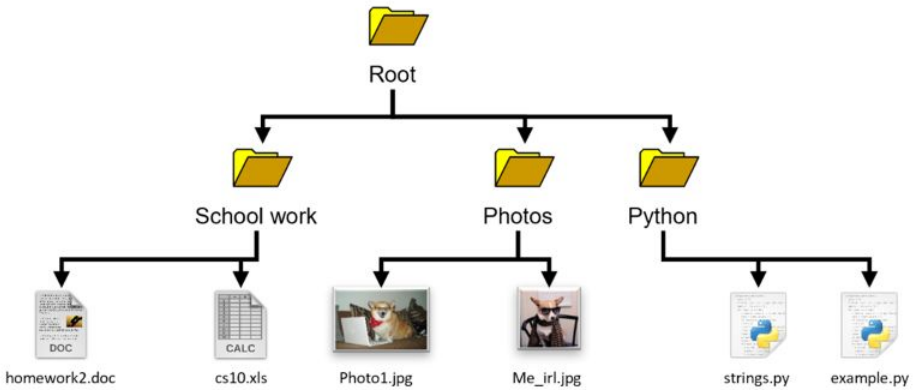
File Systems

- Your computer has a file system that lets you save and load files.
- A *file* is a sequence of data. You create files all the time, and they're organized by your computer's file system.



File Systems

The diagram can also be written as a hierarchy of text.



=

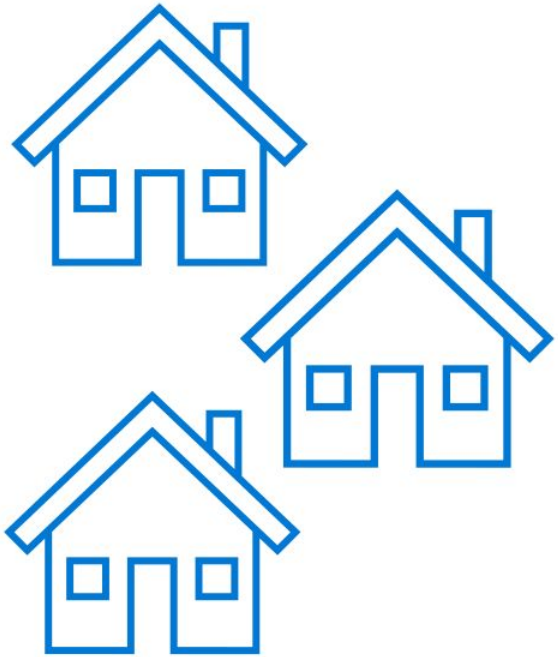
```
School work/
    homework2.doc
    cs10.xls
Photos/
    Photo1.jpg
    Me_irl.jpg
Python/
    strings.py
    example.py
```

Directories

- *Directories*, or folders, are a way to group files and other directories.
  - **Directories**: A collection of files and directories
  - **Files**: A sequence of data external to a program

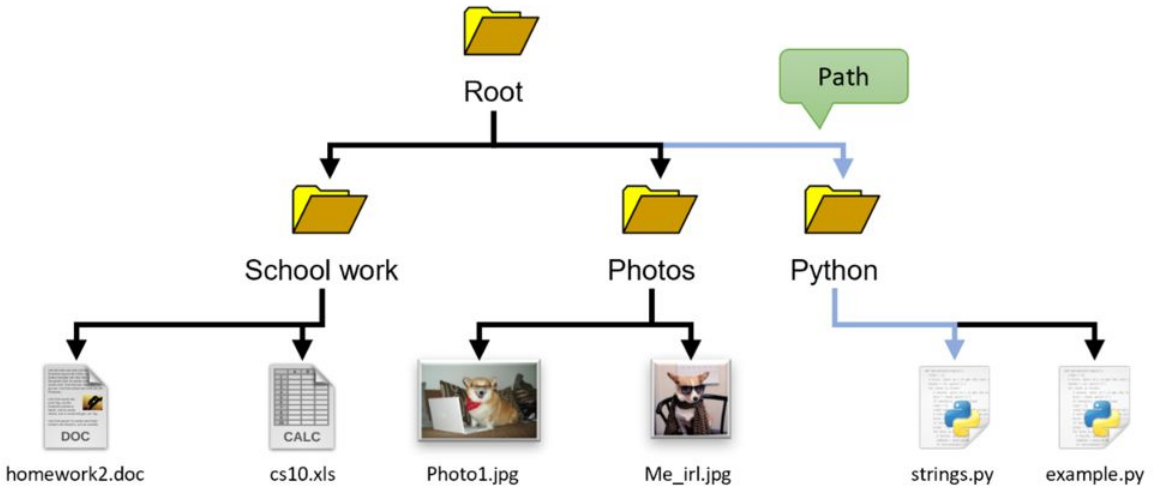
- Because you can put directories inside directories, you end up with a hierarchical system.

- Think of files like a house: Directories are neighborhoods, cities, states, and successively bigger ways to group houses.



Directories

- Each file has a unique address within your file system called a *path*.
- Use these paths to navigate files and directories like you'd navigate between houses and neighborhoods.

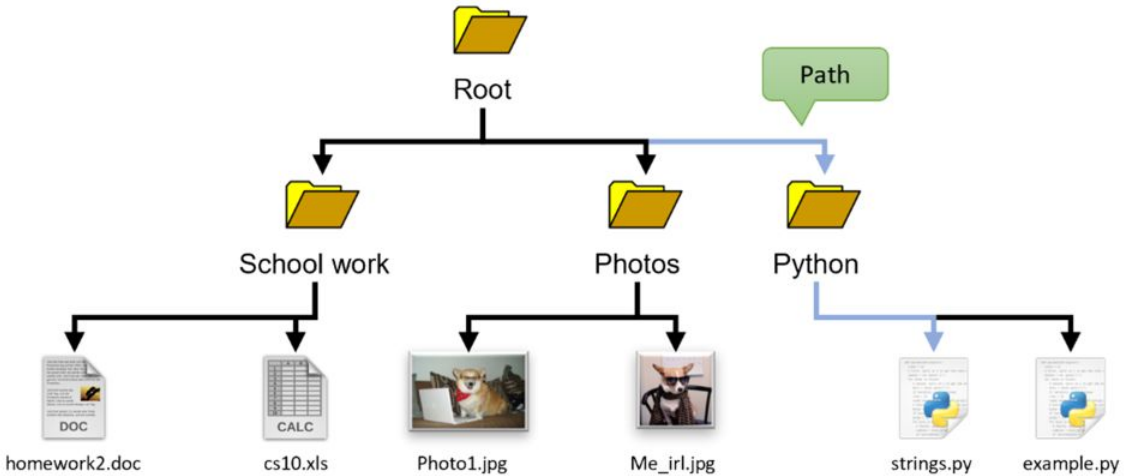


Check Your Understanding

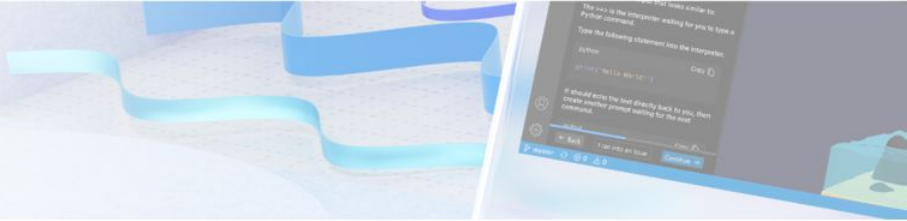
Question 1

How many directories and files are in the diagram?

- ❑ 4 directories, 6 files
- ❑ 3 directories, 7 files
- ❑ 6 directories, 4 files



Check Your Understanding



Question 1

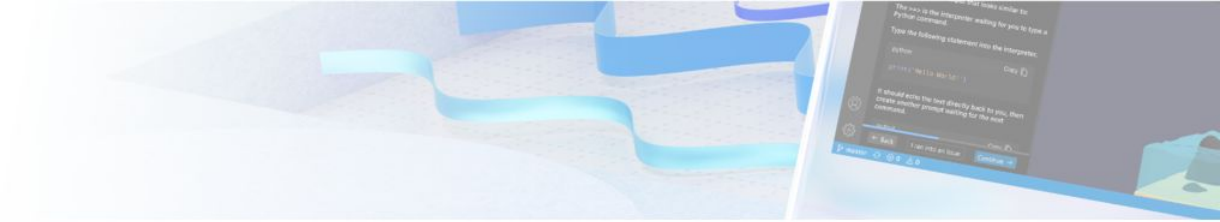
How many directories and files are in the diagram?

- ✓ 4 directories, 6 files
  The top-level directory ("Root") contains three directories. Each folders contains 2 files.

- ❑ 3 directories, 7 files

- ❑ 6 directories, 4 files

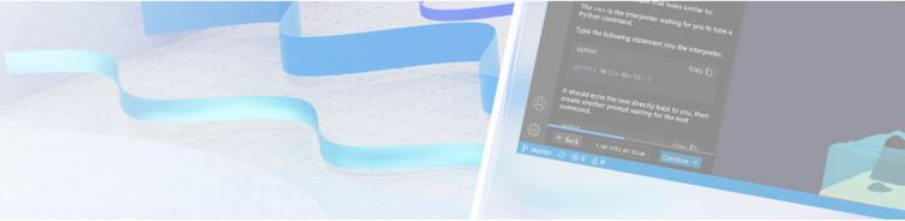
Absolute Path



```
/root/python/example.py
/home/acbart/python/example.py
C:/Users/acbart/python/example.py
C:\\Users\\acbart\\python\\example.py
```

- The full address for a file is its *absolute path*.
- Paths are usually a series of folder names separated by slashes.
- In Python, use forward slashes (/) because they work regardless of the platform.
- Otherwise, you need to escape the backslashes with extra backslashes.

Check Your Understanding



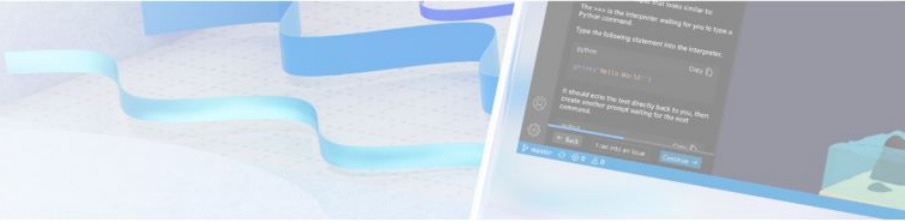
Question 2

In what directory of the following absolute paths is the `example.py` file?

```
"/root/python/example.py
/home/acbart/python/example.py
C:/Users/acbart/python/example.py
C:\\Users\\acbart\\python\\example.py
```

- ❑ `example.py`
- ❑ `python`
- ❑ `acbart`

Check Your Understanding



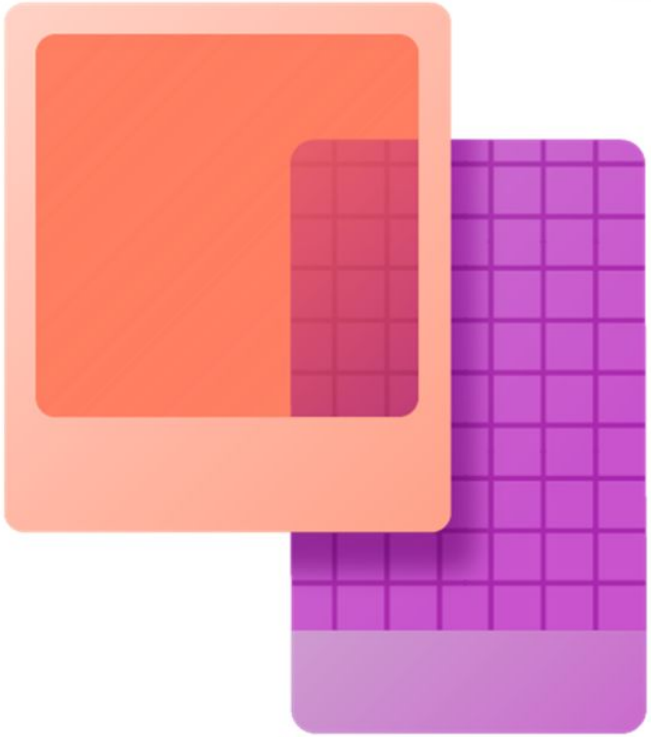
Question 2

In what directory of the following absolute paths is the `example.py` file?

```
"/root/python/example.py
/home/acbart/python/example.py
C:/Users/acbart/python/example.py
C:\\Users\\acbart\\python\\example.py
```

- ❑ `example.py`
- ✓ `Python`
  When written on the terminal, the `pwd` command can be used to print the current working directory.
- ❑ `acbart`

Terminal vs. the REPL



- Most modern programming environments (like Visual Studio Code) have terminals (command lines).
- Python provides a special command line called REPL (Read-Evaluate-Print-Loop).
- You can use REPL to interactively run Python statements and see the output.
- When using REPL, you're restricted to using regular Python language statements.

Terminal vs. REPL

- The most common terminals support their own special set of commands for interacting with the file system:
  - `pwd`: Print the current working directory.
  - `ls`: List files in the current working directory.
  - `ls` path: List files in the directory of the path.
  - `cd path`: Change the current working directory to the given path.

Terminal vs. REPL

- The Python terminal starts with three angle brackets `>>>`.
- The regular terminal can start with various kinds of symbols, such as `$>`.

The Working Directory

- A regular terminal usually has a *current working directory*.
- In Visual Studio Code, you can check your current directory by using the `pwd` command with an exclamation mark in front of it.
- PWD stands for *print working directory*.
- This works only in the Thonny command line.

```
$> pwd
'C:/Users/acbart/projects/pythonmisc'
```

Relative Paths

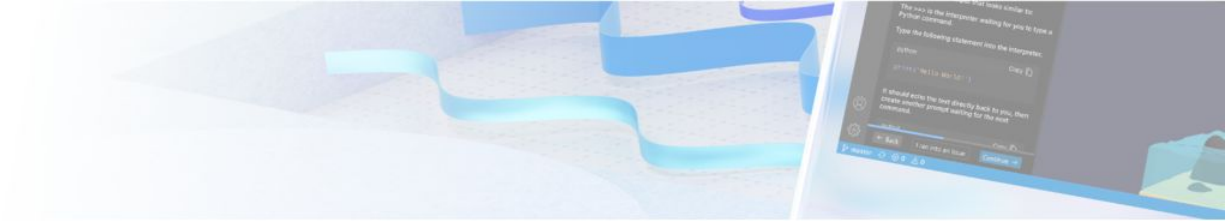
- An *absolute path* works only on your own computer.
- It's better to use a *relative path* to make your program easier to share with other people.
- A relative path is the part of an absolute path that omits the current working directory.

Relative Paths

- A relative path is usually much shorter than an absolute path.

- If your current working directory has folders, you can reference them by name without writing the absolute path.

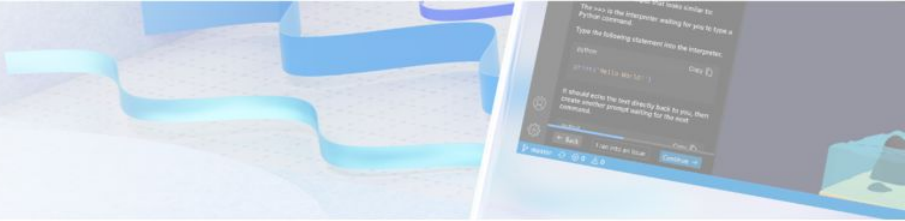
Relative Paths



```
$> ls

Date        Time        Type  Size     Name
09/17/2017  09:41 PM    <DIR>          .
09/17/2017  09:41 PM    <DIR>          ..
09/12/2017  03:11 PM    <DIR>          photos
09/14/2017  05:34 PM    <DIR>          music
09/11/2017  02:45 PM    <DIR>          python
09/02/2016  06:08 PM             6,228 homework5.pdf
09/02/2016  06:08 PM             9,650 example.py
08/08/2015  07:43 PM           340,489 historyOfDogs.png
08/08/2015  07:44 PM           422,316 taxReturns.docx
```

- You can see a list of the folders in your current directory by using the `ls` command.
- LS stands for list, as in list the files.

Check Your Understanding

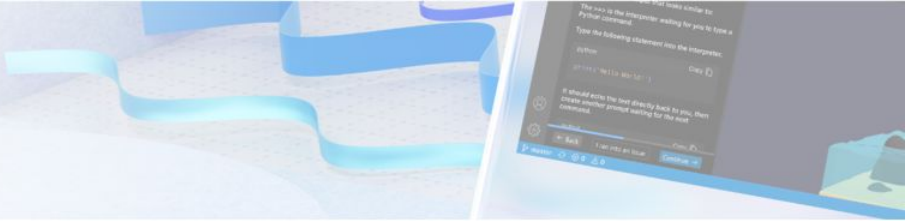


Question 4

Which command prints out the files in the directory named “homework” in the current working directory?

- ❑ `ls /homework/`
- ❑ `ls`
- ❑ `ls homework`

Check Your Understanding

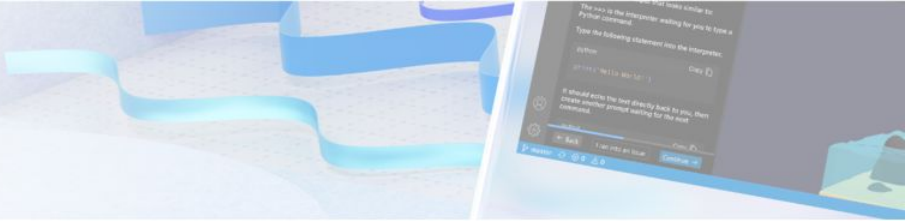


Question 4

Which command prints out the files in the directory named “homework” in the current working directory?

- ❑ `ls /homework/`
- ❑ `ls`
- ✓ `ls homework`
  This command prints out the files in a local folder named homework.

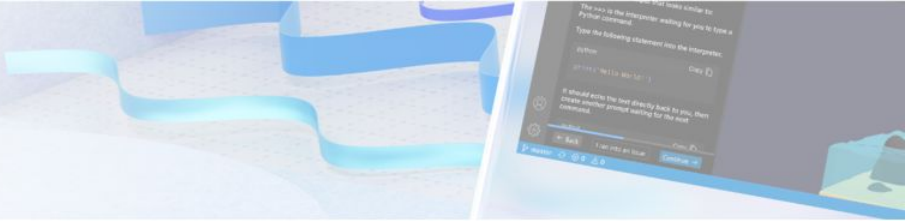
Moving Between Directories



- Use the `cd` command ("change directories") to move between directories.
- You can move to an absolute path or a relative path.
- You can move up a folder level by using a pair of periods (`..`).
- The folder above the current folder is called the *parent folder* in the path.

```
# Absolute path:
# Move to specific folder anywhere in your file system
$> cd /c/Users/acbart/projects

# Relative path:
# Move to folder in current directory
$> cd pythonmisc/

# Move up a level
$> cd ..
```

Check Your Understanding

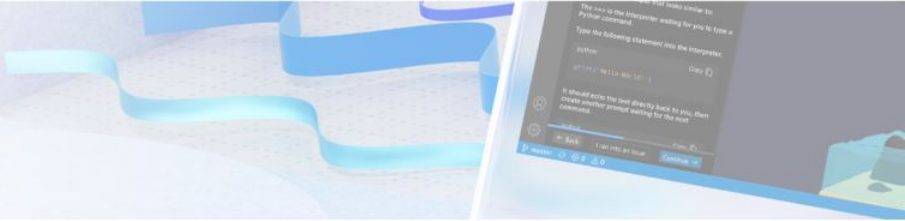


Question 5

What command moves up a directory?

- ❑ `cd ..`
- ❑ `cd`
- ❑ `cd “up”`

Check Your Understanding



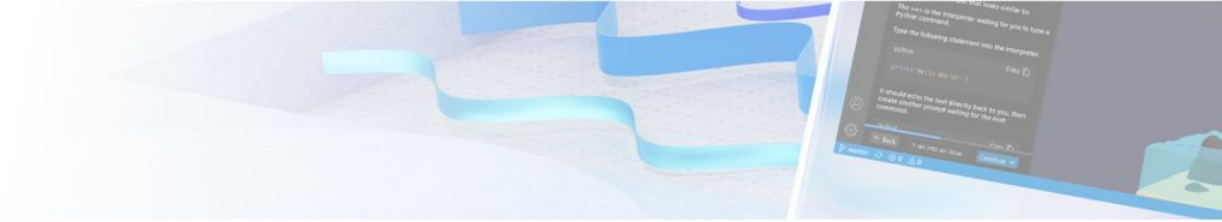
Question 5

What command moves up a directory?

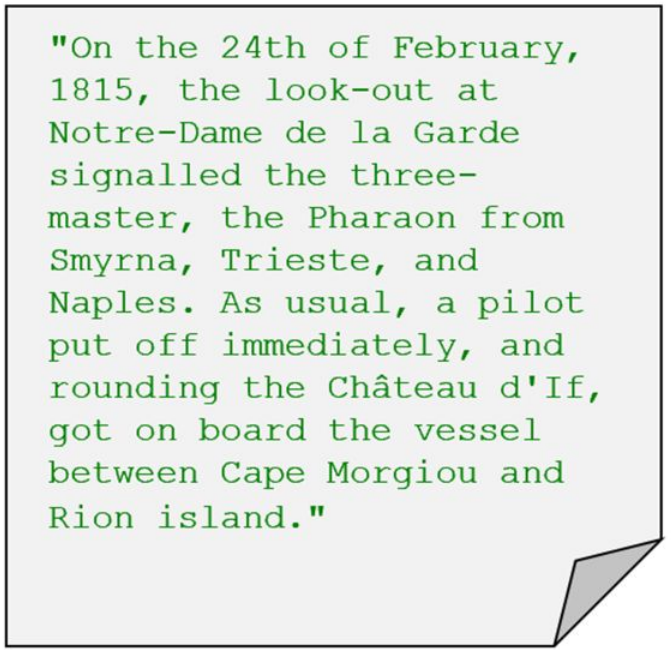
- ✓ `cd ..`
  This command moves the current working directory up a level in the path.
- ❑ `cd`
- ❑ `cd “up”`

File I/O

Files



- Files are strings of data.
- If you know the path and filename of the file, you can use Python to access it.
- You can then process the data as if it were a string or a list of strings.

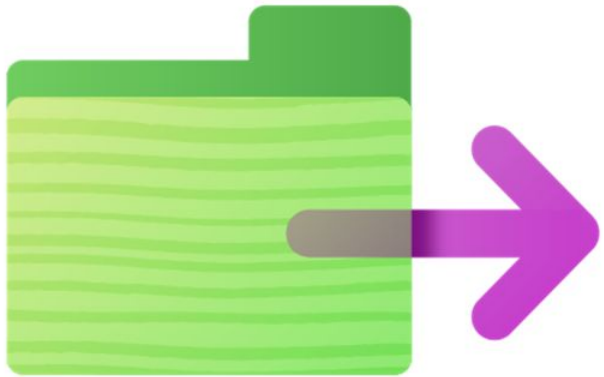


Count-of-monte-cristo.txt

Files

- The `open` function consumes the path to the file as a string and returns a `File` object.
- Typically, you store this `File` object in a variable.
- **Example**: Until you tell Python to read data from the file, the only information you have is that the file is open and ready.
- The `File` object isn't the same thing as the data inside the file.

```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

# Boring!
print(book_file)
```

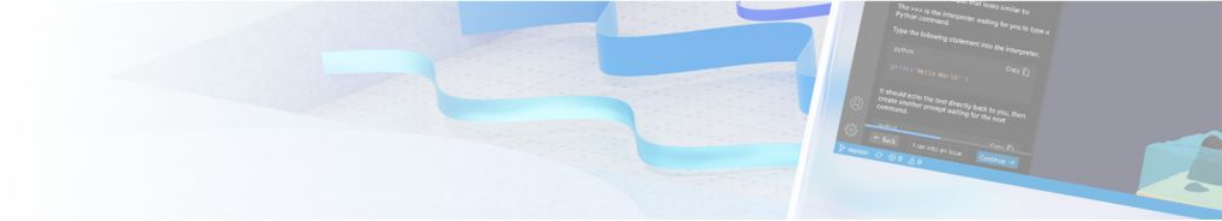


Reading Characters

```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

# Use the read() method to get the file as a string
book_text = book_file.read()
print(book_text)
```

- Get data from a `File` object by using the `.read()` method, which returns the file contents as a string.
- **Example**: open the file, read the `File` object, and then print the file's text.
- This is a multi-step process:
  1. Use the path to open the file.
  2. Read from that open file.

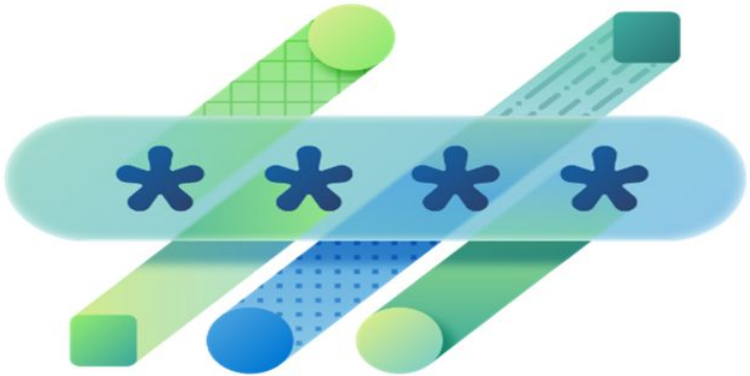
Reading Characters



- With the string loaded from the file, you can process the file character by character.
- **Example**: Open the file again and count the number of characters by using the loop pattern.

```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

# Use the read() method to get the file as a string
book_text = book_file.read()
count = 0
for character in book_text:
    count += 1
print(count)
```

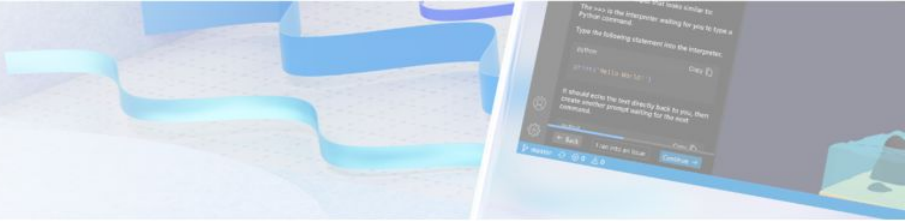
Line-by-Line File Iteration



```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

for line in book_file:
    print(line)
```

- Because a `File` is a sequence of strings (each separated by a new line), you can process it by using a `for` loop.
- **Example**: Process the file line by line.
- Using the `for` loop, you no longer need to use the `.read()` method.
- Combining the `read` and `for` loop results in string iteration.

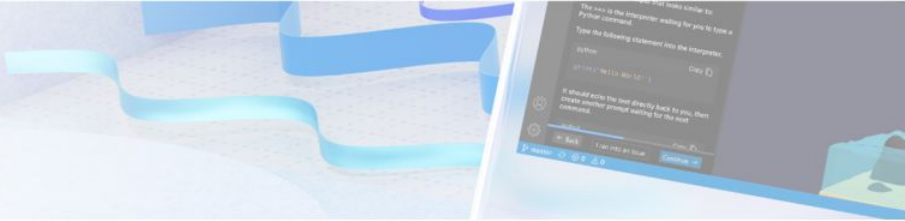
Line-by-Line File Iteration



- You can break up lines of a file by using new line characters (usually `\n`).
- When a file is read line by line, the new line characters are included in each line.
- Use the `strip` method to remove the extra whitespace from the end of the line.

```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

for line in book_file:
    print(line)
```

Check Your Understanding

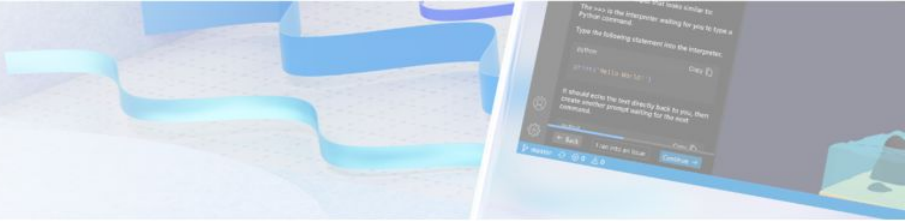


Question 1

The `open` function consumes a string representing a path. What does the `open` function return?

- ❑ A string representing the file contents.
- ❑ A list of strings representing the file contents.
- ❑ A `File` object that you can use to access the file contents.

Check Your Understanding



Question 1

The `open` function consumes a string representing a path. What does the `open` function return?

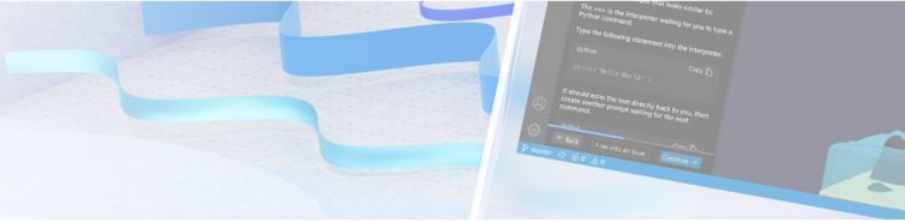
- ❑ A string representing the file contents.
- ❑ A list of strings representing the file contents.
- ✓ A `File` object that you can use to access the file contents.
  The open function returns a `File` object, which is a new type of value with special methods for accessing the actual file data.

Closing Files

- When you're done with a file, close it by using the `close` method.
- Forgetting to close a file can leak memory resources on older devices and possibly cause data loss.
- The `close` indicates to anyone reading the program that the file-reading phase is finished.
- After a file is closed, you can't use the `read` method on the file or iterate through it with a `for` loop.

```
book_path = "Count-of-monte-cristo.txt"
book_file = open(book_path)

print(book_file.read())

# This is critical!!!
book_file.close()
```

Check Your Understanding



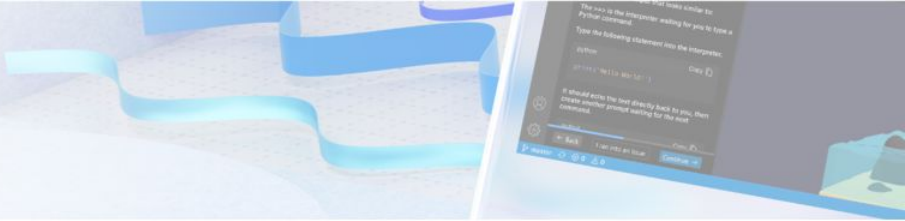
Question 2

Given a file named `grades.txt` with the contents: `90, 85, 100` and the following Python code, what is the type of `grade_data`?

```
grade_file = open("grades.txt")
grade_data = grade_file.read()
```

- ❑ `list[int]`
- ❑ `str`
- ❑ `list[str]`

Check Your Understanding



Question 2

Given a file named `grades.txt` with the contents: `90, 85, 100` and the following Python code, what is the type of `grade_data`?

```
grade_file = open("grades.txt")
grade_data = grade_file.read()
```

- ❑ `list[int]`

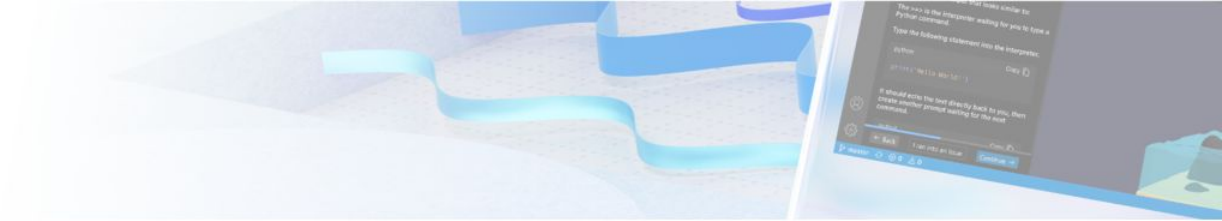
- ✓ `Str`
  The `read` method always returns a string no matter what the contents of the file are (even if the file might look like a list or integers.)

- ❑ `list[str]`

File Objects

- When you call the `open` function from before, you're given a `File` object.
- The `File` type has its own unique methods (`read`, `close`) and can be iterated by using a `for` loop.
- You can't use operators like addition (`+`) or subscripting (square brackets like `[` with an index).

File Objects



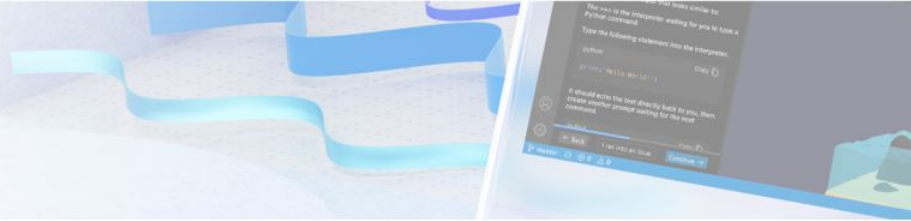
Python has many special built-in type values. For now, just remember the operations and methods for files:

- The `open` function that takes a string path and returns an open `File` object
- The `close` method of `File` objects that frees up the resource
- The `read` method of `File` objects that returns the contents of the file as a string
- The `for` loop iteration over the `File` object as a sequence of strings (separated by new lines)

FileNotFoundError

- File systems are tricky because everyone has a different setup.
- When you try to open a file that doesn't exist, Python raises a `FileNotFoundError` and suggests that the file doesn't exist.
- Ask yourself:
  - Do I have the right file name?
  - Do I have the right path?
  - Is the file where I think it is?
  - Is my program where I think it is?

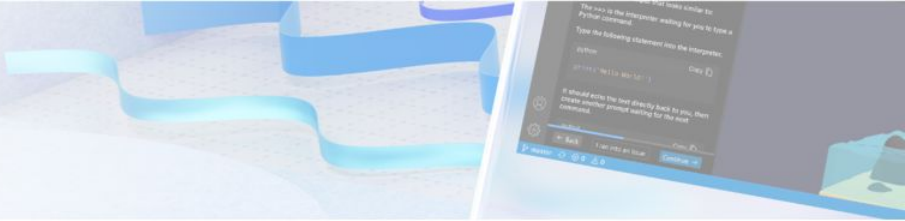
Example File Processing



```
score_sum = 0
data_file = open('scores.txt')

for line in data_file:
    score_sum = score_sum + int(line.strip())

data_file.close()
print(score_sum)
```

- **Example**: Process a list of numbers in a file. Each number represents a score.
- The code shows the sum pattern to add each of the scores together.
- Strip off the new lines at the end of each line, then convert that line to a number. When you read data from a file, it comes in as a string.

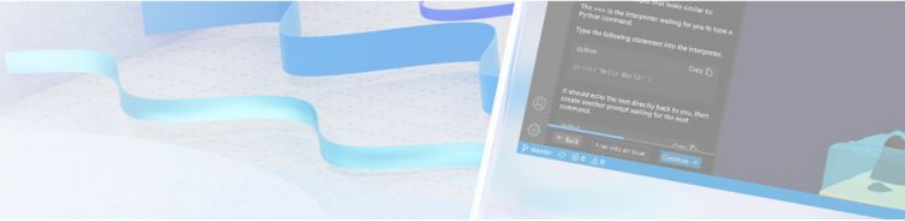
Example File Processing



- Even if a file contains only numbers, the values returned by the `read` method and line-by-line iteration will still just be strings.
- Strings can contain numeric characters, but that doesn't make those values integers.
- Until you explicitly convert the contents by using the `int` or `float` function, you have string values.

```
score_sum = 0
data_file = open('scores.txt')

for line in data_file:
    score_sum = score_sum + int(line.strip())

data_file.close()
print(score_sum)
```

Check Your Understanding

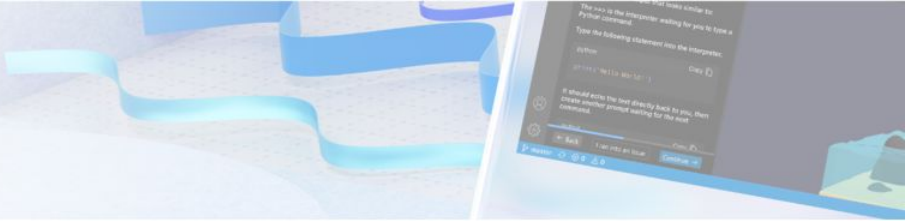


Question 3

When you iterate through a `File` object with a `for` loop, how do you go through the file?

- ❑ Character by character
- ❑ Sentence by sentence
- ❑ Line by line

Check Your Understanding



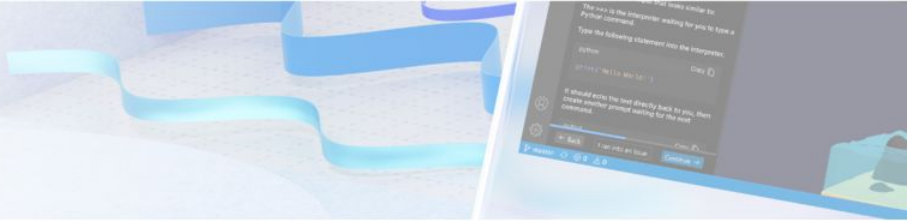
Question 3

When you iterate through a `File` object with a `for` loop, how do you go through the file?

- ❑ Character by character
- ❑ Sentence by sentence
- ✓ Line by line

  A file is organized into lines separated by new line characters (`\n`). When you iterate through the file with a `for` loop, you get each line as a string value (including the new line.)

Check Your Understanding

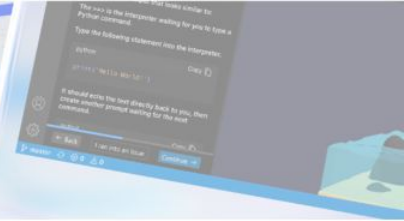


Question 4

When should you close a file?

- ❑ Immediately after opening the file
- ❑ After you finish reading the file
- ❑ At the very end of the program, on the last line

Check Your Understanding



Question 4

When should you close a file?

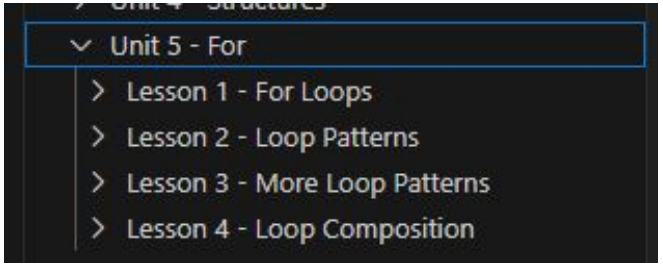
- ❑ Immediately after opening the file
- ✓ After you finish reading the file
  Once a file has been read, you have no further use for it. That's the best time to close the file.
- ❑ At the very end of the program, on the last line

In-Class Lab

Lab - GitHub Module

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

- **VSCode Unit 6: Lesson 1 - Lesson 4**



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.

Next Week

Next week will entail:

- Introduction to OOP
- Data visualization
- Measures of dispersion
- Conda installation

Jupyter: scratchpad of the data scientist

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