Digital Career Institute

Python Course: Input & Output

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help students learn different methods of data input and output. By the end of this submodule, the learners will be able to understand:

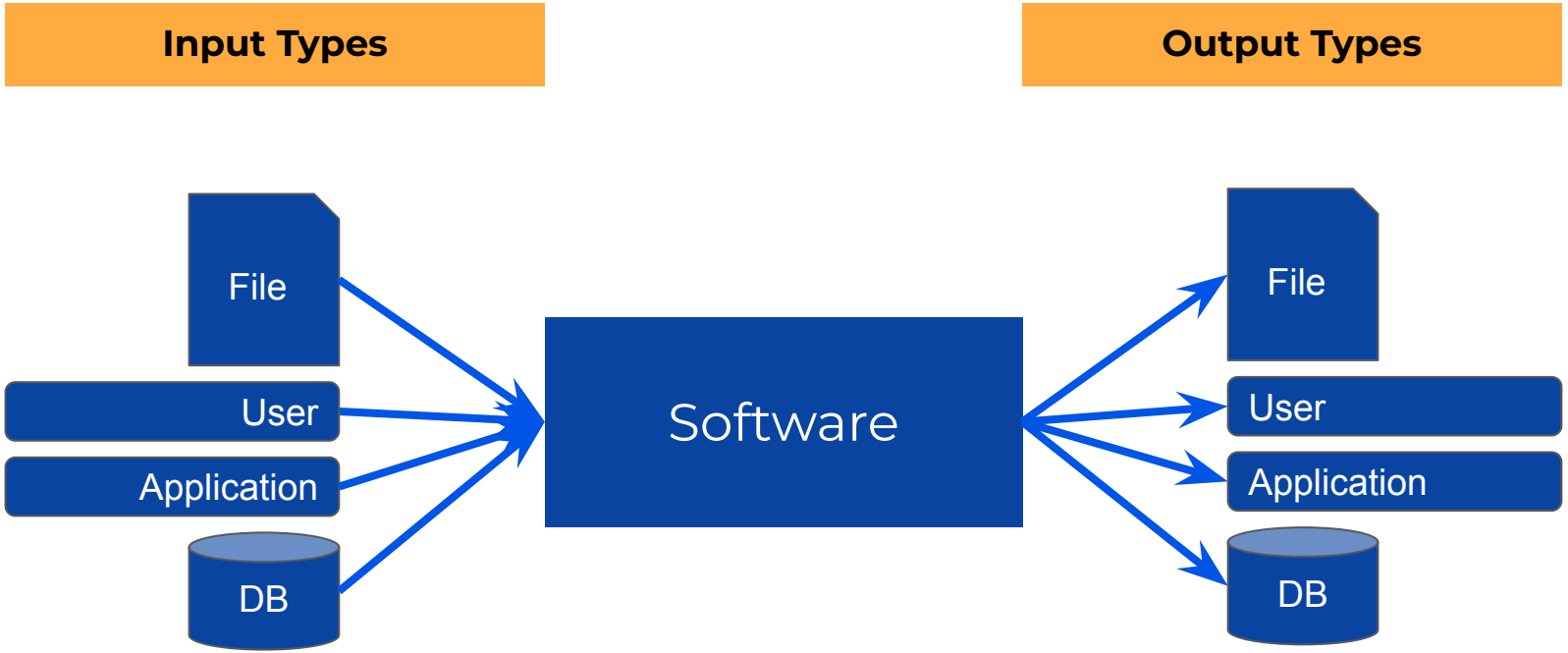
- How to input/output data to a text and binary file.
- The difference between the **`bytes`** and **`bytearray`** types.
- What are I/O streams and most of their methods and properties.
- How to work with directories and files.
- Different ways to capture user input and output data to the user.

Topics

- The **`open`** function and reading a file.
- The **`with`** operator.
- Writing files.
- File opening modes.
- Bytes-like objects and binary files.
- Streams.
- Using the file system.
- User I/O.
- Application I/O.

Input & Output

Data Input & Output



File I/O: open()

```
>>> file = open("the_hobbit.txt")
```

The **open** function is a built-in Python function. It does not need to be imported from any package.

```
>>> print(open.__doc__)
```

Opening Files: path

```
>>> file = open("the_hobbit.txt")
```

```
>>> file = open("books/the_hobbit.txt")
```

```
>>> file = open("/home/user/the_hobbit.txt")
```

We can use **relative** and **absolute** paths.

File Input

Reading Text Files

```
>>> file = open("the_hobbit.txt")
>>> print(file)
<_io.TextIOWrapper name='test.txt'
mode='r' encoding='UTF-8'>
```

The **open** function returns a **File Object**.

Reading Text Files: read()

```
>>> file = open("the_hobbit.txt")
>>> print(file.read())
In a hole in the ground there lived a
hobbit...
```

The **`file`** object has a **`read`** method to access the contents of a text file.

Reading Text Files: read()

```
>>> file = open("the_hobbit.txt")
>>> print(file.read(10))
In a hole
```

The **`read`** method has a parameter **`size`** to limit the length of the reading.

Defaults to `EOF` (End Of File).

Reading Text Files: readlines()

```
>>> file = open("the_hobbit.txt")
>>> for line in file.readlines():
...     print(line)
In a hole in the ground there lived a
hobbit.

Not a nasty, dirty, wet hole, filled...
```

The **`file`** object has a **`readlines`** method to access the contents of a text file as a **list of string lines**.

Iterating Lines

Digital Career Institute

The **`file`** object
is an **iterable**.

```
>>> for line in file.readlines():
...     print(line)
```

=

```
>>> for line in file:
...     print(line)
```

```
>>> lines = file.readlines():
```

=

```
>>> lines = list(file)
```

Reading Text Files: readline()

```
>>> file = open("the_hobbit.txt")
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>>
```

The **`file`** object has a **`readline`** method to access the contents of a **single line** in a text file.

`readline` also has a parameter **`size`** that defaults to **`EOL`** (End Of Line).

Reading Text Files: The Position

```
>>> file = open("the_hobbit.txt")
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>> print(file.readline())
Not a nasty, dirty, wet hole, filled
with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole
with nothing in it to sit down on or to
eat: it was a hobbit-hole, and that
means comfort.
```

Repeating again the call to `readline` will return the **next line**.

Python keeps track of the **position** of the reading.

We can only read forward!

Reading Text Files: tell()

```
>>> file = open("the_hobbit.txt")
>>> print(file.tell())
0
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>> print(file.tell())
46
```

The **`file`** object has a method **`tell`** that returns the current **position**.

Reading Text Files: seek()

```
>>> file = open("the_hobbit.txt")
>>> print(file.tell())
0
>>> file.seek(46)
>>> print(file.tell())
46
>>> print(file.read(41))
Not a nasty, dirty, wet hole, filled
with
>>> print(file.tell())
87
```

The **`file`** object has a method **`seek`** that changes the current **position**.

Closing Files: closed()

```
>>> file = open("the_hobbit.txt")
>>> print(file.closed)
False
>>> file.close()
>>> print(file.closed)
True
>>> print(file.read())
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    print(file.fileno())
ValueError: I/O operation on closed file
```

Files must **always** be closed when we don't need them any more.

The `file` object has a method `close` to close the file.

The file object also has a property `closed` that returns `True` if the file has been closed, and `False` otherwise.

Closing Files: with

```
>>> with open("the_hobbit.txt") as file:
...     print(file.closed)
...
False
>>> print(file.closed)
True
```

The `with` statement will automatically close the file, **even if there is an error exception** occurring in the code.

It is always preferable to open files with the `with` statement.

File Output

File Output: write()

```
>>> file.write("Some text")
```

The **write** method writes some text into the file.

Writing Text Files

```
>>> with open("todo_list.txt") as file:
...     file.write("My ToDos' Header")
...
Traceback (most recent call last):
  File "test.py", line 18, in <module>
    file.write("My ToDos' Header")
io.UnsupportedOperation: not writable
```

Writing to files is, by default, **denied**.

The **open** function, by default, opens files for **reading** purposes **only**.

Opening Mode

The **open** function has a **mode** parameter to indicate which mode should be used when opening the file.

The default value of **mode** *is "read a text file".*

```
>>> file = open("the_hobbit.txt")
>>> print(file)
<_io.TextIOWrapper name='test.txt'
mode='r' encoding='UTF-8'>
```

Opening Mode: Reading

```
>>> file = open("the_hobbit.txt")
```

=

```
>>> file = open("the_hobbit.txt", mode="r")
```

Opening a text file for **reading** purposes.

Opening Mode: Writing

```
>>> file = open("the_hobbit.txt", mode="w")
```

Opening a text file for **writing** purposes.

Writing Files

```
>>> with open("todo_list.txt", "w") as file:
...     file.write("My ToDos' Header")
...
>>>
```

todo_list.txt

```
My ToDos' Header
```

Writing Files: writelines()

```
>>> to_dos = ["Go shopping\n", "Call mum\n"]
>>> with open("todo_list.txt", "w") as file:
...     file.write("My ToDos:")
...     file.write("\n\n")
...     file.writelines(to_dos)
...
>>>
```

todo_list.txt

```
My ToDos':

Go shopping
Call mum
```

The methods `write` and `writeline` do not add a line break automatically. In this example it is added with `\n`.

Other Opening Modes

Opening Modes

```
>>> file = open("todo_list.txt", mode="r")
```

Opening Modes	
r	Reading only
w	Writing only (overwriting)
x	Creating only (fails if already exists)
a	Appending only

r, **w** and **a** accept a modifier **+** to update the file.

Opening Modes

Digital Career Institute

Properties by mode	r	r+	w	w+	a	a+
Can read	✓	✓		✓		✓
Can write		✓	✓	✓	✓	✓
Can write after seek		✓	✓	✓		
Can create			✓	✓	✓	✓
Can truncate			✓	✓		
Starting position	start	start	start	start	end	end

*To **truncate** is to remove all the content of the file without deleting the file.*

Opening Modes

Rules of thumb

r	Only reading is required.
w	Only writing is required.
w+	Both reading and writing are required.

Creating Files

The `x` mode is used if we only want to write the file when it does not exist.

```
>>> with open("todo_list.txt", mode="x") as file:
...     file.write("My ToDos:")
```

If the file exists raises an exception

Overwrites or creates the file

```
>>> with open("todo_list.txt", mode="w") as file:
...     file.write("My ToDos:")
```

Writing Files

The `r+` will let us overwrite without truncating.

todo_list.txt

```
My ToDos:

Go shopping
Call mum
```

```
>>> with open("todo_list.txt", "r+") as file:
...     file.seek(26)
...     file.write("Something else")
...
>>>
```

todo_list.txt

```
My ToDos:

Go shopping
Something else
```

Writing Files

The **a** *mode will start at the end of the file.*

```
>>> with open("todo_list.txt", "a") as file:
...     print(file.tell())
35
...     file.seek(0)
...     print(file.tell())
0
...     file.write("Something else\n")
...     print(file.tell())
49
>>>
```

todo_list.txt

```
My ToDos:

Go shopping
Call mum
Something else
```

Binary Files

The open() Modes

```
>>> file = open("the_hobbit.txt")
```

=

```
>>> file = open("the_hobbit.txt", mode="r")
```

=

```
>>> file = open("the_hobbit.txt", mode="rt")
```

Format Mode

Opening Mode

Opening a **t**ext file for **r**eading.

Format Modes

```
>>> file = open("test.txt", mode="rt")
```

Opening Modes	
r	Reading only
w	Writing (overwriting)
x	Creating only (fails if already exists)
a	Appending

Format Modes	
t	Text (default)
b	Binary

Text and binary files have the same properties and methods, but they are created differently.

Writing Binary Files

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = "In a hole in the ground
there lived a hobbit.")
...     file.write(content)
...
Traceback (most recent call last):
  File "test.py", line 24, in <module>
    file.write(content)
TypeError: a bytes-like object is required,
not 'str'
```

Normal text (of type **str**) cannot be written into binary files.

The file in binary mode requires a **byte-like object**.

Byte-like Objects

Digital Career Institute

bytes & **bytearray**

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = bytes("In a hole in the ground
there lived a hobbit.\nNot a nasty, dirty, wet
hole, filled with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole with
nothing in it to sit down on or to eat: it was a
hobbit-hole, and that means comfort.", "utf-8")
...     file.write(content)
...
>>>
```

The function **bytes** can also be used to convert the string.

The type **bytes** is the equivalent of the type **str** for binary data.

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = "In a hole in the ground there
lived a hobbit.\nNot a nasty, dirty, wet hole,
filled with the ends of worms and an oozy smell,
nor yet a dry, bare, sandy hole with nothing in it
to sit down on or to eat: it was a hobbit-hole,
and that means comfort.".encode("utf-8")
...     file.write(content)
...
>>>
```

The method **`encode`** can also be used to convert to **`bytes`**.

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = b"In a hole in the ground
there lived a hobbit.\nNot a nasty, dirty, wet
hole, filled with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole with
nothing in it to sit down on or to eat: it was a
hobbit-hole, and that means comfort."
...     file.write(content)
...
>>>
```

Prepending `b` to the string also works.

Bytes to Strings

```
>>> content = b"Hello."
>>> print(type(content))
<class 'bytes'>
>>> print(type(content.decode("utf-8")))
<class 'str'>
```

The reverse (converting a **`bytes`** object to **`str`**) can be done with the **`decode`** method.

Using the Type bytes

```
>>> content = "Hello."
>>> print(content[0])
H
>>> content = b"Hello"
>>> print(content[0])
72
```

The type **`bytes`**, like **`str`**, is a sequence.

Each item corresponds to a letter but, in this case, is represented as a decimal integer.

Using the Type bytes

```
>>> content = "Hello."
>>> print(len(content))
5
>>> content = b"Hello"
>>> print(len(content))
5
>>> content[0] = "Y"
```

Error

As sequences, they both can use similar functions.

The **`bytes`** type is also immutable.

The Type bytearray

A **`bytearray`** *is a mutable* **`bytes`**.

```
>>> content = bytearray("Hello", "utf-8")
>>> print(type(content))
<class 'bytearray'>
```

The type **`bytearray`** is the equivalent of the type **`list`** for binary data.

Using the Type bytearray

```
>>> content = bytearray(b"Hello")
>>> print(content.decode("utf-8"))
Hello
>>> content[0:1] = b"Y"
>>> content.append(ord("w"))
>>> print(content.decode("utf-8"))
Yellow
```

Like in a **`list`**, we can change one of the items by specifying an index or using slicing.

`ord` returns the decimal that represents the letter **`w`** and is being appended to the **`bytearray`**.

Summary of Data Types

	Immutable	Mutable
text	str	list
binary	bytes	bytearray

Streams & the io Module

Streams

Streams are a more generic term for the **`File Object`** mentioned earlier.

Streams

Digital Career Institute

Features

permissions	`read-only`, `write-only`, `read-write`	**sequence**	They have an order.
seekable	The position can be detected and changed.	**position**	There is a position.

The io Module

The `io` module provides classes to work with **streams**.

The io Module

StringIO & BytesIO

```
>>> import io
>>> data = io.StringIO("Hello World!")
>>> print(data.read(5))
Hello
```

```
>>> import io
>>> data = io.BytesIO(b"Hello World!")
>>> print(data.read(5))
b'Hello'
```

Stream Methods & Properties

Digital Career Institute

Properties

closed

`True` if file is closed.

mode

Mode parameter.

encoding

Encoding the file is using.

name

Name of the file.
Only when opening files.

Stream Methods & Properties

Digital Career Institute

Methods

read()	Read all the file.	**readlines()**	Return a list of read lines.
readline()	Read a single line.	**write()**	Write text into the file.

Stream Methods & Properties

Digital Career Institute

Methods

writelines()	Writes a list of lines.	**close()**	Closes the file.
tell()	Returns the position.	**seek()**	Changes the position.

Stream Methods & Properties

Digital Career Institute

Methods

readable() `True` if the file can be read.

writable() `True` if the file can be written.

seekable() `True` if the file can be seeked.

truncate() Delete the content of the file.

We learned ...

- That we can use the built-in function **`open`** to work with files.
- That we need to override the **`mode`** parameter if we want to do something different than **`reading-only`**.
- How to work with binary files and what is the difference between the types **`bytes`** and **`bytearray`**.
- What are I/O streams.
- That streams have a **`position`** that we can **`seek`** and **`tell`** and we can use to overwrite parts of a text.

Using the File System

Using the File System

When files are being created ...

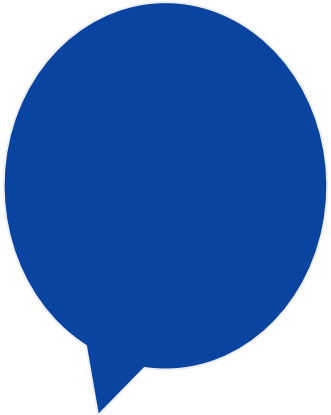
Does the file already exist?

Does the directory exist?

What other files are in the directory?

Using the File System

When files are being created ...



Create a directory

Delete a directory

Explore a directory

Delete a file

These are all operations on **paths**.

The pathlib Module

Digital Career Institute

The **`pathlib`** module is a utility library that provides access to various operations with paths in the file system.

Create a File

Digital Career Institute

paths.py

```
from pathlib import Path


file = Path("/home/PythonCourse/text.txt")
file.open("w").write("My notes")
```

Files can also be created using **`pathlib`**.

The **`Path`** object has a method **`open`** that returns a stream.

Delete a File

Digital Career Institute

paths.py

```
from pathlib import Path


file = Path("/home/PythonCourse/test.py")
file.unlink()
```

A directory cannot be removed this way.

```
$ python3 paths.py
```

Delete a File

Digital Career Institute

paths.py

```
from pathlib import Path


file = Path("/does/not/exist.py")
file.unlink()
```

By default, the **`unlink`** method will raise an exception if the file does not exist.

```
$ python3 paths.py
FileNotFoundError: [Errno 2] No such file or directory:
'does/not/exist.py'
```

Delete a File

Digital Career Institute

paths.py

```
from pathlib import Path


file = Path("/does/not/exist.py")
file.unlink(missing_ok=True)
```

The **missing_ok** argument will prevent the exception if it is set to **True**.

```
$ python3 paths.py
$
```

Python 3.8

Remove a Directory

paths.py

```
from pathlib import Path


directory = Path("/home/PythonCourse/test/")
directory.rmdir()
```

A directory **must be empty** before it can be deleted.

```
$ python3 paths.py
```

Create a Directory

Digital Career Institute

paths.py

```
from pathlib import Path


directory = Path("test")
directory.mkdir()
```

```
$ python3 paths.py
```

Get the Current Directory

Digital Career Institute

/home/PythonCourse/paths.py

```
from pathlib import Path


print(Path.cwd())
```

```
$ python3 paths.py
/home/PythonCourse
```

Rename a File or Directory

Digital Career Institute

paths.py

```
from pathlib import Path


file = Path("foo.txt")
file.open("w").write("Some text")
new_file = Path("bar.txt")
file.replace(new_file)
print(new_file.open().read())
```

```
$ python3 paths.py
Some text
```

List the Directory Content

paths.py

```
from pathlib import Path


books = Path("books")
for item in books.iterdir():
    print(item)
```

```
$ python3 paths.py
the_hobbit.bin
the_hobbit.txtt
todo_list.txt
io
```

Search a Directory

paths.py

```
from pathlib import Path


books = Path("books")
for path in books.glob("*.txt"):
    print(path)
```

```
$ python3 paths.py
books/the_hobbit.txt
books/dracula.txt
books/frankenstein.txt
$
```

glob returns any path object (files and directories) matching the indicated pattern.

Search a Directory

Digital Career Institute

paths.py

```
from pathlib import Path


books = Path("books")
for path in books.glob("*/"):
    print(path)
```

```
$ python3 paths.py
books/fantasy
books/horror
books/biography
$
```

glob can be used to match subdirectories.

Search a Directory

paths.py

```
from pathlib import Path


books = Path("books")
for path in books.glob("**/*.txt"):
    print(path)
```

```
$ python3 paths.py
books/frankenstein.txt
books/fantasy/the_hobbit.txt
books/horror/dracula.txt
$
```

`glob` can also be used recursively.

Get the File Path

paths.py

```
from pathlib import Path


print(__file__)
file = Path(__file__)
print(file.resolve())
```

`__file__` is a reference to the current file.

`resolve` returns the full path of the file.

```
$ python3 paths.py
paths.py
/home/DCI/PythonCourse/paths.py
```

Get the File's Directory Path

paths.py

```
from pathlib import Path


file_path = Path("the_hobbit.txt")
print(file_path.parent)
print(file_path.resolve().parent)
print(file_path.parent.resolve())
```

`parent` returns the parent directory of the provided path.

If the **`file_path`** is relative **`resolve`** can be used either before or after using **`parent`**.

```
$ python3 paths.py
.
/home/DCI/PythonCourse
/home/DCI/PythonCourse
```

Get the Object Name

paths.py

```
from pathlib import Path


file = Path("/home/DCI/main.py")
print(file.name)
directory = Path("/home/DCI")
print(directory.name)
```

name returns the name of the object.

```
$ python3 paths.py
main.py
DCI
```

Join Paths

paths.py

```
from pathlib import Path


home = Path("/home")
user = "DCI"
course = "PythonCourse"
path = home.joinpath(user, course)
print(path)
```

The **`joinpath`** method will return a new path merging the inputs into the original path.

```
$ python3 paths.py
/home/DCI/PythonCourse
```

Join Paths

paths.py

```
from pathlib import Path


home = Path("/home")
user = "DCI"
course = "PythonCourse"
path = home / user / course
print(path)
```

The `/` operator and the `joinpath` method serve the same purpose.

```
$ python3 paths.py
/home/DCI/PythonCourse
```

Existence of a Path

paths.py

```
from pathlib import Path


path = Path("/home/DCI/PythonCourse/")
print(path.exists())
```

We can use either a file path or a directory path.

```
$ python3 paths.py
True
```

The Pythonic Way

Digital Career Institute

Ask **forgiveness**,
not permission.

In Python, Exceptions have a very small cost in performance. Smaller than most operations.

Existence of a Path: Asking Forgiveness

Digital Career Institute

paths.py

```
from pathlib import Path


path = Path("/path/")
if path.exists():
    file = path.open()
else:
    print("Does not exist")
```

Asking permission

path_example.py

```
from pathlib import Path


path = Path("/path/")
try:
    file = path.open()
except FileNotFoundError:
    print("Does not exist")
```

Asking forgiveness

Nature of a Path

paths.py

```
from pathlib import Path


path = Path("/home/DCI/PythonCourse/")
print(path.is_absolute())
print(Path("the_hobbit.txt").is_absolute())
```

`is_absolute` returns **`True`** if the path is **absolute**.

```
$ python3 paths.py
True
False
```

Nature of a Path

paths.py

```
from pathlib import Path


path = Path("/home/DCI/PythonCourse/")
print(path.is_file())
```

`is_file` returns **`True`** if the path is a **file**.

```
$ python3 paths.py
False
```

Nature of a Path

paths.py

```
from pathlib import Path


path = Path("/home/DCI/PythonCourse/")
print(path.is_dir())
```

`is_dir` returns **`True`**
if the path is a **directory**.

```
$ python3 paths.py
True
```

The os Module

Digital Career Institute

The `os` module is a utility interface to various operations that concern the **operating system** where the code is being executed.

The `os` module can perform many operations on the operating system, such as getting information, working with environment variables or executing system processes.

It also has some feature to work with paths.

Walk the Directory Tree

os_example.py

```
import os
from pathlib import Path


for dir_name, subdirs, files in os.walk(Path.cwd()):
    print("*" * 20)
    print("Directory:", dir_name)
    print("Subdirectories:", subdirs)
    print("Files:", files)
```

`walk` loops through all the directory tree and returns every directory, its subdirectories and its files.

Walk the Directory Tree

os_example.py

```
import

for dir
    pri
    pri
    pri
    pri
```

```
$ python3 os_example.py
********************
Directory: /home/DCI/PythonCourse
Subdirectories: ['io']
Files: ['the_hobbit.bin', 'the_hobbit.txt', 'todo_list.txt']
********************
Directory: /home/DCI/PythonCourse/io
Subdirectories: []
Files: ['test.py']
```

The os.path Module

Digital Career Institute

The **`os.path`** module is a utility interface to various operations that concern the **File System** where the code is being executed.

This module is scarcely used and it is being replaced by **`pathlib`**, but it has some additional features.

Properties of a Path

Digital Career Institute

path_example.py

```
import os


print(os.path.getsize("the_hobbit.txt"))
print(os.path.getmtime("the_hobbit.txt"))
```

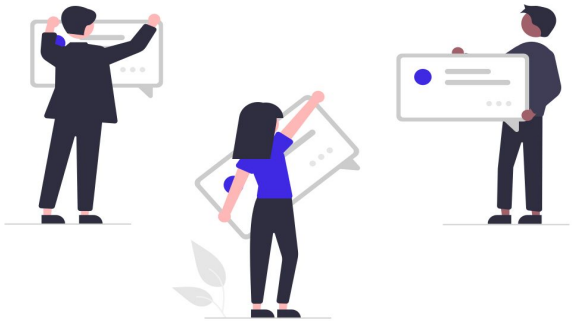
getsize returns the size of a path.

getmtime returns the timestamp of the last modification.

```
$ python3 path_example.py
247
1629893207.7288237
```

We learned ...

- That we can use the **`pathlib`** module to create, delete and explore directories or delete files.
- That **`pathlib`** also provides information about paths, their nature and their properties.
- That it is better to catch the **`FileNotFoundError`** exception than using **`os.path.exists`** to prevent it.
- That the **`os`** module provides additional features to work with paths and the operating system.



Reflection Round

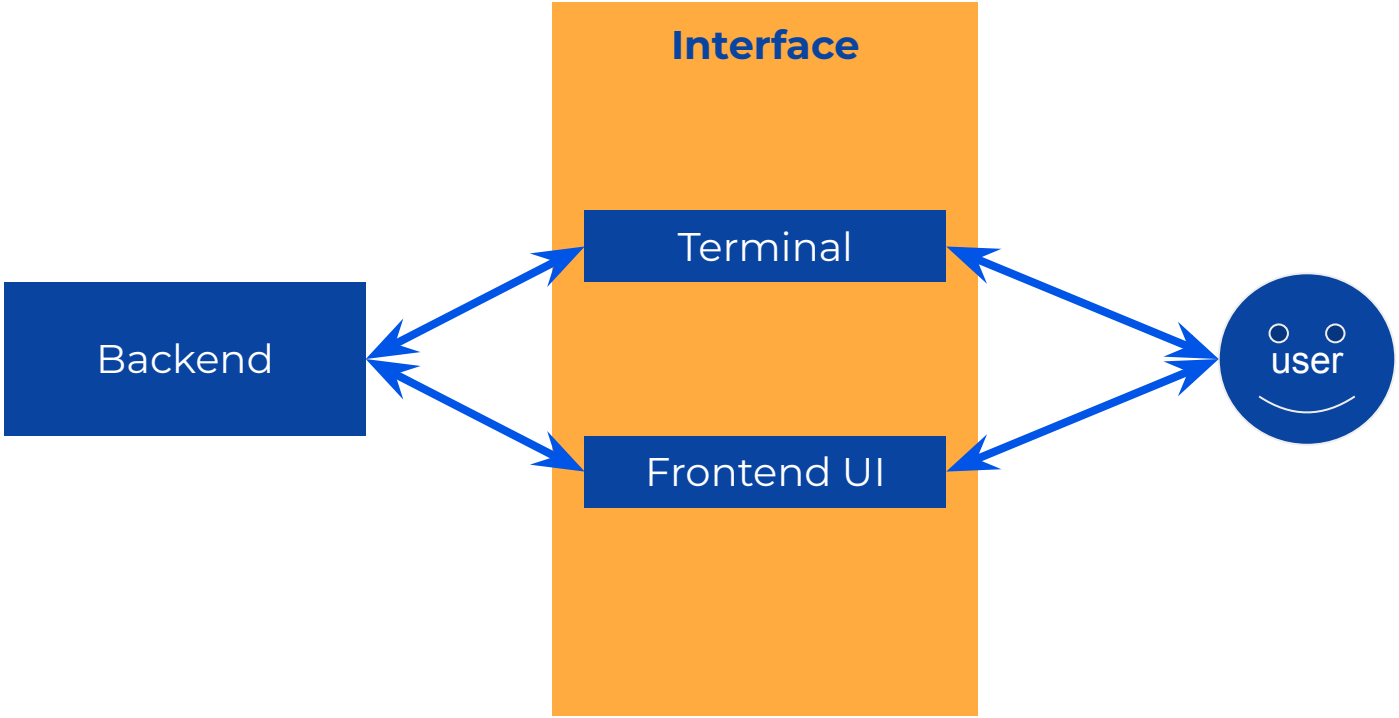
Digital Career Institute

Suggested Topics

- Discuss advantages and disadvantages of storing data in files rather than in the database.
- Name 10 possible uses of a file based approach.
- Discuss the convenience (or not) of using IO streams to manipulate simple text. Compare it to using simple `str` objects.

User I/O

User Input & Output



Command Line I/O

```
>>> name = input("What is your name? ")
What is your name? Anakin
>>> print(f"Hello {name}!")
Hello Anakin!
```

User
Input

Terminal
Output

Command Line Output: Print

Digital Career Institute

```
>>> print("Hello", "Anakin!")
Hello Anakin!
```

```
>>> print("H", "e", "l", "l", "o", "A",
...       "n", "a", "k", "i", "n", "!")
...
H e l l o A n a k i n !
```

The **`print`** function accepts any number of **positional** arguments.

Command Line Output: Print

```
>>> print("H", "e", "l", "l", "o", "A",
...       "n", "a", "k", "i", "n", "!",
...       , sep="")
...
HelloAnakin!
```

The **`print`** function has a **`sep`** keyword argument that takes a string.

The **`sep`** argument is used as a separator **between** positional arguments.

Defaults to a white space: `  `.

Command Line Output: Print

```
>>> def greet():
...     print("Hello", end=" ")
...     print("Anakin!")
...
>>> greet()
Hello Anakin!
```

Defaults to a line break: `\n`.

The `print` function has an `end` keyword argument that takes a string.

The `end` argument is printed **after** all the positional arguments.

Command Line Output: Print

```
>>> with open("todo_list.txt", "w") as file:
...     print("My ToDos:", file=file)
...
>>>
```

The **`print`** function has a **`file`** keyword argument that takes a **`File Object`**.

If a **`file`** is present, it will print to a file (not binary).

Defaults to **`sys.stdout`**.

todo_list.txt

```
My ToDos:
```

Command Line Output: Print

```
>>> import time
>>> num_seconds = 3
>>> counts = reversed(range(num_seconds + 1))
>>> for countdown in counts:
...     if countdown > 0:
...         print(countdown, end='...')
...         time.sleep(1)
...     else:
...         print('Go!')
...
```

The **`print`** function uses a buffer, which may produce unexpected behavior.

This code prints a countdown, but it prints it all at once, after 3 seconds.

Command Line Output: Print

Digital Career Institute

```
>>> import time
>>> num_seconds = 3
>>> counts = reversed(range(num_seconds + 1))
>>> for countdown in counts:
...     if countdown > 0:
...         print(countdown, end='...',
...               flush=True)
...         time.sleep(1)
...     else:
...         print('Go!')
...
```

The **`print`** function has a **`flush`** keyword argument that takes a **`Boolean`**.

If **`flush = True`**, it will empty the buffer and print every second.

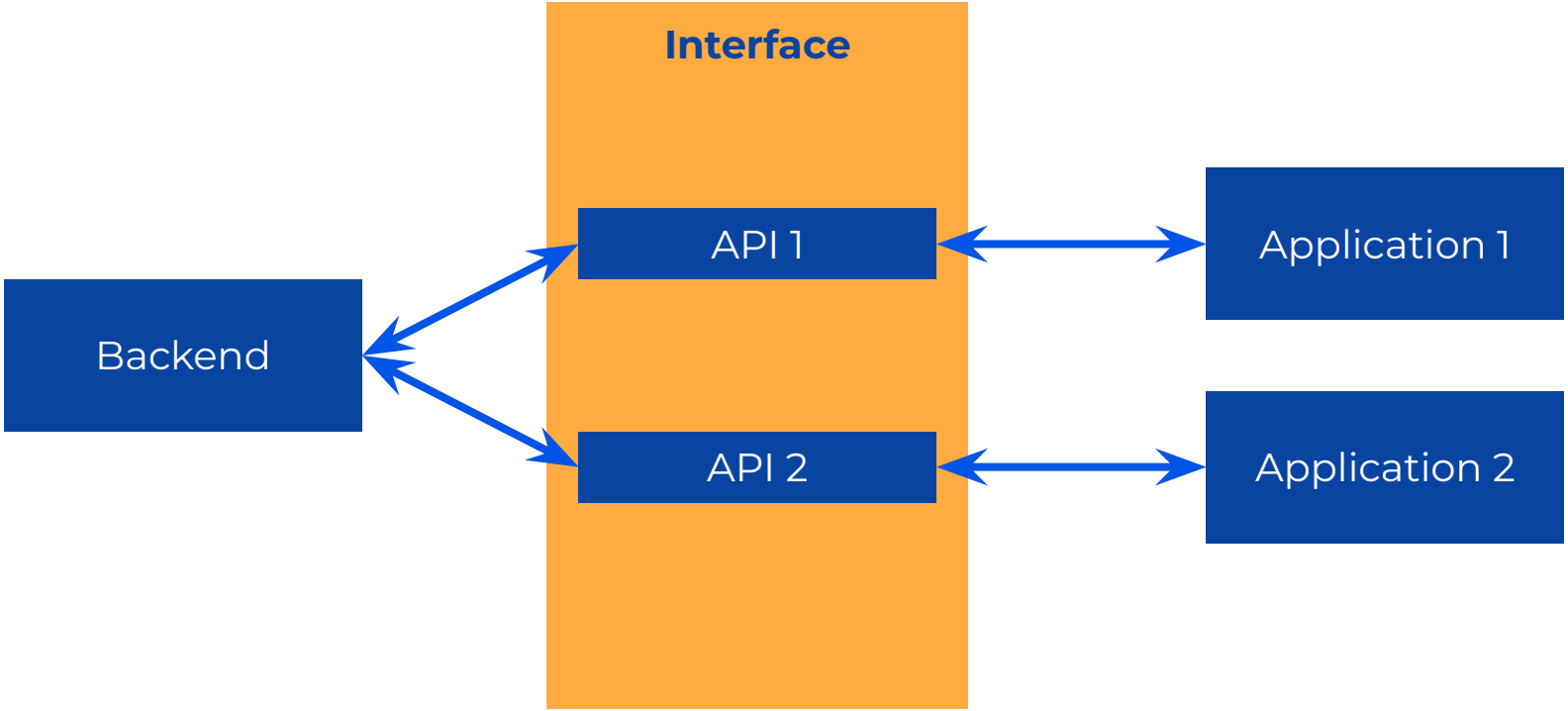
Print a Pretty Dictionary

```
>>> import json
>>> data = {'username': 'jdoe',
...         'password': 's3cret'}
...
>>> pretty = json.dumps(data, indent=4,
...                       sort_keys=True)
...
>>> print(pretty)
{
    "password": "s3cret",
    "username": "jdoe"
}
```

The **`json.dumps`** method has **`indent`** and **`sort_keys`** keyword arguments.

Application I/O

Application Input & Output



API Interaction Example

```
>>> import requests
>>> response = requests.get(
...     url="https://reqres.in/api/users"
... )
>>> print(response.content)
b'{"page":1,"per_page":6,"total":12,"tota
l_pages":2,"data":[...
```

API Input

Screen Output

API Interaction Example

```
>>> import requests
>>> data = {
...     "name" = input("Type your name"),
...     "age" = input("Type your age ")
... )
>>> response = requests.post(
...     data=data,
...     url="https://reqres.in/api/users"
... )
```

User Input

API Output

We learned ...

- That we can obtain user input from the command line or from a frontend interface.
- That we can customize the way the **`print`** function works with keyword parameters.
- That we can use the **`print`** function to output data into a file.
- That the **`print`** function is buffered and how to flush the buffer.
- That our software can also input and output data from/to another application.

Documentation

Documentation

- File I/O
  https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
  https://www.tutorialspoint.com/python/python_files_io.htm
  https://www.w3schools.com/python/python_file_handling.asp
  https://www.w3schools.com/python/python_file_open.asp
  https://www.w3schools.com/python/python_file_write.asp
- Bytes-like objects
  https://www.w3resource.com/python/python-bytes.php
  https://docs.python.org/3/library/stdtypes.html#bytes
  https://docs.python.org/3/library/stdtypes.html#bytearray
  https://www.python.org/dev/peps/pep-0257/

Documentation

- Streams
  https://docs.python.org/3/library/io.html
- File system
  https://docs.python.org/3/library/os.html#module-os
  https://docs.python.org/3/library/os.path.html#module-os.path
- User I/O
  https://docs.python.org/3/library/functions.html#print
- Application I/O
  https://pypi.org/project/requests/

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute