Command-Line Arguments

Command-line arguments allow you to pass information to your Python script when you execute it from the terminal. This makes your scripts more flexible and reusable, as you don't need to modify the code for different inputs.

Accessing Command-Line Arguments

The **sys module** provides access to command-line arguments through sys.argv. sys.argv is a list where:

- sys.argv[0] is the name of the script itself.
- sys.argv[1] is the first argument, sys.argv[2] the second, and so on.

Example:

Let's create a script named greet.py:

Python

```
# greet.py

import sys

if len(sys.argv) > 1:

   name = sys.argv[1]

   print(f"Hello, {name}!")

else:

   print("Please provide a name as an argument.")
```

Implementation:

To run this script from your terminal:

Bash

```
python greet.py Alice
```

Output:

```
Hello, Alice!
```

Parsing Command-Line Arguments

For more complex scenarios where you need different types of arguments (e.g., flags, optional arguments), manually parsing sys.argv can become cumbersome. This is where dedicated argument parsing modules come in handy.

Argparse Module for More Complex Argument Parsing

The **argparse module** is the standard library for parsing command-line arguments in Python. It simplifies the process of defining, parsing, and validating arguments, including positional arguments, optional arguments, and flags.

Key Concepts:

- **ArgumentParser object:** The main object that holds all the information necessary to parse the command line.
- **add_argument() method:** Used to define individual command-line arguments. You can specify argument names, types, default values, help messages, and whether they are required.
- **parse_args() method:** Parses the arguments from the command line and returns an object containing the argument values.

Example:

Let's create a script calculator.py that takes two numbers and an operation:

Python

```
# calculator.py

import argparse


parser = argparse.ArgumentParser(description="A simple calculator program.")

parser.add_argument("num1", type=float, help="The first number")

parser.add_argument("num2", type=float, help="The second number")

parser.add_argument("-o", "--operation", choices=["add", "subtract", "multiply", "divide"],

                default="add", help="The operation to perform (default: add)")
```

```
args = parser.parse_args()


result = 0
if args.operation == "add":
    result = args.num1 + args.num2
elif args.operation == "subtract":
    result = args.num1 - args.num2
elif args.operation == "multiply":
    result = args.num1 * args.num2
elif args.operation == "divide":
    if args.num2 != 0:
        result = args.num1 / args.num2
    else:
        print("Error: Cannot divide by zero!")
        exit(1)


print(f"Result: {result}")
```

Implementation:

Bash

```
python calculator.py 10 5 --operation add
```

Output:

```
Result: 15.0
```

Bash

```
python calculator.py 10 2 -o divide
```

Output:

```
Result: 5.0
```

Bash

```
python calculator.py --help
```

Output (partial):

```
usage: calculator.py [-h] [-o {add,subtract,multiply,divide}] num1 num2


A simple calculator program.


positional arguments:
  num1             The first number
  num2             The second number


options:
  -h, --help       show this help message and exit
  -o {add,subtract,multiply,divide}, --operation {add,subtract,multiply,divide}
                 The operation to perform (default: add)
```

Working with Modules

Modules are fundamental to organizing and reusing code in Python. They allow you to break down large programs into smaller, manageable, and reusable files.

Creating and Importing Modules

A **module** is simply a Python file (.py) containing Python definitions and statements.

Creating a Module:

Let's create a module named my_utils.py:

Python

```
# my_utils.py

def add(a, b):
    return a + b


def subtract(a, b):
    return a - b


PI = 3.14159
```

Importing Modules:

You can import modules into other Python scripts using the import statement.

- **import module_name**: Imports the entire module. You access its contents using module_name.item.
- **from module_name import item1, item2**: Imports specific items (functions, variables, classes) from a module directly into your current namespace.
- **from module_name import ***: Imports all items from a module directly into your current namespace. **(Generally discouraged in larger projects due to potential name collisions.)**
- **import module_name as alias**: Imports the module and gives it an alias.

Example Usage in main_script.py:

Python

```
# main_script.py

import my_utils as mu

from my_utils import PI

result_add = mu.add(10, 5)

result_subtract = mu.subtract(20, 7)

print(f"Addition result: {result_add}")

print(f"Subtraction result: {result_subtract}")

print(f"Value of PI: {PI}")
```

Implementation:

Save both my_utils.py and main_script.py in the same directory. Then run main_script.py:

Bash

```
python main_script.py
```

Output:

```
Addition result: 15

Subtraction result: 13

Value of PI: 3.14159
```

Packages and the Module Search Path

As your projects grow, you'll likely want to organize related modules into **packages**. A package is a directory containing a special file named __init__.py (which can be empty). This __init__.py file tells Python that the directory should be treated as a package.

Creating a Package:

Let's create a package named math_operations:

```
my_project/
 ├── main_app.py
└── math_operations/
    ├── __init__.py
    ├── basic_ops.py
   └── advanced_ops.py
```

math_operations/basic_ops.py:

Python

```
# basic_ops.py
def add(a, b):
    return a + b


def subtract(a, b):
    return a - b
```

math_operations/advanced_ops.py:

Python

```python
# advanced_ops.py

def power(base, exp):

    return base ** exp

def factorial(n):

    if n == 0:

        return 1

    else:

        return n * factorial(n-1)
```

Importing from Packages:

You can import modules or specific items from modules within a package using dot notation.

Example in main_app.py:

Python

```python
# main_app.py

from math_operations import basic_ops

from math_operations.advanced_ops import factorial

sum_result = basic_ops.add(15, 8)

fact_result = factorial(5)

print(f"Sum: {sum_result}")

print(f"Factorial of 5: {fact_result}")
```

Implementation:

Run main_app.py:

Bash

```bash
python main_app.py
```

Output:

Sum: 23

Factorial of 5: 120

The Module Search Path:

When you import a module, Python searches for it in a specific order of directories. This order is defined by sys.path.

1. **The directory containing the input script:** If you run python my_script.py, the directory where my_script.py resides is searched first.
2. **PYTHONPATH environment variable:** A list of directories specified by the user.
3. **Standard library directories:** Directories where Python's built-in modules are installed.
4. **Site-packages directories:** Directories where third-party packages are installed (e.g., via pip).

You can inspect sys.path to see the current search path:

Python

```
import sys

print(sys.path)
```

Regular Expressions (Regex) 🔍

Regular expressions are powerful sequences of characters that define a search pattern. They are invaluable for pattern matching, searching, and replacing text within strings. Python's built-in **re module** provides full support for regular expressions.

Pattern Matching Using Regular Expressions

The re module offers several functions for pattern matching:

- **re.match(pattern, string)**: Attempts to match a pattern at the **beginning** of a string. Returns a match object if successful, None otherwise.
- **re.search(pattern, string)**: Scans through a string looking for the first location where the pattern produces a match. Returns a match object if successful, None otherwise.
- **re.findall(pattern, string)**: Returns a list of all non-overlapping matches of the pattern in the string.

- **re.finditer(pattern, string)**: Returns an iterator yielding match objects for all non-overlapping matches.

Common Regex Metacharacters and Sequences:

Character	Description	Example Pattern	Matches
.	Any character (except newline)	a.b	acb, a$b, a1b
^	Start of the string	^Hello	Hello World
$	End of the string	World$	Hello World
*	Zero or more occurrences of the preceding character/group	ab*c	ac, abc, abbbc
+	One or more occurrences of the preceding character/group	ab+c	abc, abbbc
?	Zero or one occurrence of the preceding character/group	ab?c	ac, abc
{m}	Exactly m occurrences	a{3}b	aaab
{m,n}	Between m and n occurrences (inclusive)	a{2,4}b	aab, aaab, aaaab
[]	Character set (matches any one character inside the brackets)	[aeiou]	a, e, i, o, u
[^]	Negated character set (matches any character NOT inside)	[^0-9]	Any non-digit
`	`	OR (matches either the expression before or after)	`cat
()	Grouping (for applying quantifiers or capturing)	(ab)+	ab, abab
\	Escape special characters, or denote special sequences	\.	A literal dot (.)
\d	Digit (0-9)	\d{3}	123, 456

Character	Description	Example Pattern	Matches
\D	Non-digit	\D	a, B, $
\w	Word character (alphanumeric + underscore)	\w+	hello_world
\W	Non-word character	\W	, !, @
\s	Whitespace character (space, tab, newline, etc.)	\s	,\t,\n
\S	Non-whitespace character	\S	a, 1, $

Export to Sheets

Example:

```
import re


text = "The quick brown fox jumps over the lazy dog. My email is example@domain.com."


# Using re.search

match = re.search(r"fox", text)

if match:

    print(f"Found 'fox' at index {match.start()} to {match.end()}")


# Using re.findall to find all numbers

numbers = re.findall(r"\d+", "There are 123 apples and 45 oranges.")

print(f"Numbers found: {numbers}")


# Finding email addresses

email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"

emails = re.findall(email_pattern, text)
```

```
print(f"Emails found: {emails}")
```

Output:

```
Found 'fox' at index 16 to 19

Numbers found: ['123', '45']

Emails found: ['example@domain.com']
```

Replacing and Searching for Patterns

The re module also provides functions for replacing parts of a string that match a pattern.

- **re.sub(pattern, replacement, string, count=0)**: Replaces all occurrences (or count occurrences) of the pattern with the replacement string.
- **re.subn(pattern, replacement, string, count=0)**: Similar to re.sub, but returns a tuple of (new_string, number_of_substitutions_made).

Example:

```
import re


sentence = "I have 3 apples and 5 oranges. My phone number is 123-456-7890."


# Replace all digits with '#'

new_sentence = re.sub(r"\d", "#", sentence)

print(f"Digits replaced: {new_sentence}")


# Replace phone number with 'REDACTED'

phone_number_pattern = r"\d{3}-\d{3}-\d{4}"

redacted_sentence = re.sub(phone_number_pattern, "REDACTED", sentence)

print(f"Phone number redacted: {redacted_sentence}")
```

Output:

```
Digits replaced: I have # apples and # oranges. My phone number is ###-###-####.

Phone number redacted: I have 3 apples and 5 oranges. My phone number is REDACTED
```