Python I

Raul P. Pelaez

February 3, 2026

Contents

1 Introduction

Today, we will cover regular expressions and error handling.

Info

Regular Expressions
A regular expression (regex or regexp for short) is a special text string for describing a search pattern. We use regular expressions to search for and match text patterns in strings. Regular expressions are widely used in programming languages and text editors for tasks like searching, replacing, and validating text. For instance, the search-and-replace function in a text editor is a simple instance of regular expressions.
The simplest form of a regular expression is a literal string, such as "Hello". This regular expression will match the string "Hello" in any text. Check out this example:

Regex example

Pattern: `Hello`
Test strings:

Hello, world! Hello again!
Hi, world!
hello, world!

Throughout the lesson, we will see our regexps in action with examples like this one, in which matches are highlighted in green. Note that, in general, regular expression matching works on a line-by-line basis.

Info

Error Handling

Inevitably, our programs will sometimes encounter errors or impossible situations. As programmers, we must make sure that our code can handle these situations gracefully. Often, there is no way to recover from an error. In these cases, we should at least provide a clear error message for the user and exit the program cleanly.

Programs can fail in many ways, such as:

- Trying to open a file that doesn't exist.
- Dividing by zero.
- An internet connection failing (timing out, URL not found, etc.).
- A user entering invalid data (e.g., entering text when a number is expected).
- Running out of memory.
- A function being called with incompatible arguments.
- A function being called with unexpected argument types.

Python provides several mechanisms for handling errors, mainly exceptions, assertions and type hints. During this lesson, we will focus on the first two.

Info

Exceptions

Exceptions are events that occur during program execution that disrupt the normal flow of instructions. Python uses exceptions to handle errors gracefully, preventing the program from crashing.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero")
except Exception as e: # Generic exception handler
    print(f"An error occurred: {e}")
else: # Executes if no exceptions are raised
    print(f"Result: {result}")
finally: # Always executes
    print("This always runs")
```

Output

```
Error: Division by zero
This always runs
```

In Python terminology, snippets of code that produce an exception when failing are said to "raise" an exception. The code that handles the exception is said to "catch" or "handle" the exception.

Info

Assertions
Assertions are used to test assumptions within the code. If an assertion fails, an `AssertionError` is raised. They are primarily for debugging purposes – checks made during development that should always be true.
Example:

```
def validate_age(age):
    # The message after the comma will be displayed if the assertion fails
    assert age >= 0, "Age cannot be negative"
    print("Age is valid")
```

Assertions are used to check that the code is working as expected. Failed assertions indicate a bug in the code. Although assertions raise an exception, it is bad practice to catch them, failed assertions are considered fatal (unrecoverable) errors. For instance in the previous example, we should consider that there is something inherently wrong with the calling code if the age sent to the function is negative.

2 Regular Expressions

Regular expressions (regex or regexp) are powerful tools for pattern matching within text. Think about it as search-and-replace on steroids.

Info

The simplest match: `in`
Sometimes, you just want to check if a string contains a specific substring. In this case, you can use the `in` operator, which is much simpler than using regex. For instance:

```
print("cat" in "The cat is black")
print("dog" in "The cat is black")
```

Output

```
True
False
```

Advice

Pandas allows the use of regexp in many of its string-handling functions, like `str.contains()`, `str.extract()`, `str.replace()`, etc. Essentially any function argument in Pandas that somehow denotes a string pattern can be a regular expression. For instance, the `pd.read_csv()` has an argument called `sep` (the pattern that is used to separate columns in the CSV file), which can be a regular expression. In this case, `sep=r"\s+"` would mean that columns are separated by one or more spaces.

Info

Raw strings
Regex syntax makes use of several special characters, like `\` or `{`, that can mean something in Python (for instance, `\` is used to escape characters, like when we want to print a newline character, `\n`). To avoid having to escape these characters, we can use raw strings. Raw strings are denoted by an `r` before the string, like `r"Hello"`. In raw strings, the backslash is treated as a literal character, not as an escape character. It is a way of telling Python "do not try to interpret any special characters in this string".

Example

```
print("Two\nlines")
print("-"*15)
print(r"Single\nline")
```

Output

```
Two
lines
---------------
Single\nline
```

Advice

Practicing regex
The website regex101.com is an excellent resource for practicing regular expressions. It provides a regex tester and debugger, allowing you to test your regex patterns against sample text. The site also explains each part of the regex pattern, making it a great learning tool. My usual workflow is to write the regex pattern in the site, test it with some sample text, and then copy it to my code.

Advanced

A (Very Brief) History of Regular Expressions
In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions. In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene's notation.
Since then, regexes have appeared in many programming languages, editors, and other tools as a means of determining whether a string matches a specified pattern. Python, Java, and Perl all support regex functionality, as do most Unix tools and many text editors.
Excerpt from Real Python.

2.1 Basic Syntax

A regular expression is a sequence of characters that defines a search pattern. Simple patterns, like `word`, match literal strings (in this case, `word`). More complex patterns use special *metacharacters* to match various character sets or positions within the text.

The following are examples of regular expressions:

- `cat`: Matches the string "cat".
- `\d{3}`: Matches any three consecutive digits.
- `^[0-9]{3}-[0-9]{2}-[0-9]{4}:\s([a-zA-Z]+)$`: Matches lines starting with a number in the format "123-45-6789: Name". Stores the name in a capture group. The name must be composed of letters only and be at the end of the line.

Usage of regular expressions normally lies in one of the following categories:

- Finding lines of text that match a pattern.
- Extracting parts of a string that match a pattern.

- Replacing/deleting parts of a string that match a pattern.

Info

Metacharacters
A metacharacter is a character (or sequence of characters) that has a special meaning in a regular expression. For instance, `\d` matches any digit, `\w` matches any alphanumeric character, and `\s` matches any whitespace character.
Often, metacharacters mean the opposite when capitalized, like `\D` (matches any non-digit), `\W` (matches any non-alphanumeric character), and `\S` (matches any non-whitespace character).

Info

Regex in Python: the `re` module
Python's built-in `re` module provides functions for working with regular expressions. Key functions include:

- `re.search()`: Searches for the first occurrence of a pattern.
- `re.findall()`: Finds all occurrences of a pattern.
- `re.sub()`: Replaces occurrences of a pattern.
- `re.compile()`: Compiles a regex pattern for efficiency (useful for repeated use of the same pattern).
- `re.match()`: Matches a pattern at the beginning of a string.

Example

```python
import re

text = "The quick brown fox jumps over the lazy dog."
pattern = r"fox"  # r"" denotes a raw string literal
match = re.search(pattern, text)
if match:
    print(f"Found '{match.group(0)}'")

matches = re.findall(r"\b\w{4}\b", text) # finds 4-letter words
print(f"Four letter words: {matches}")

new_text = re.sub(r"fox", "cat", text)
print(f"New text: {new_text}")

# Compile for efficiency
compiled_pattern = re.compile(r"\b\w{4}\b")
matches2 = compiled_pattern.findall(text)
print(f"Four letter words (compiled): {matches2}")
```

Output

```
Found 'fox'
Four letter words: ['over', 'lazy']
New text: The quick brown cat jumps over the lazy dog.
Four letter words (compiled): ['over', 'lazy']
```

Check the Python regex documentation for more information. The regexp section in Real Python is also a great guided resource.

Advice

Regex flavors
Regular expressions are not unique to Python. They are a standard feature in many programming languages and tools. However, the syntax and features of regular expressions can slightly vary between different implementations. For instance, the regex syntax in Python is similar to that in Perl, but not identical. If you are familiar with regex in one language, you may need to adjust your patterns when working in another language.
You can find a detailed comparison of flavors here, and the Python regex documentation here.

2.2 Metacharacters

Info

Metacharacter cheatsheet

- `.`: Matches any single character (except newline).
- `*`: Matches zero or more occurrences of the preceding character.
- `+`: Matches one or more occurrences of the preceding character.
- `?`: Matches zero or one occurrence of the preceding character.
- `{n}`: Matches exactly `n` occurrences of the preceding character.
- `[]`: Defines a character set. For example, `[abc]` matches 'a', 'b', or 'c'. `[a-z]` matches any lowercase letter.
- `[^]`: Negates a character set. For example, `[^abc]` matches any character except 'a', 'b', or 'c'.
- `^`: Matches the beginning of a string. `$` Matches the end of a string.
- `\`: Escapes special characters. For example, `\.` matches a literal dot.
- `\d`: Matches any digit. `\D` Matches any non-digit.
- `\w`: Matches any alphanumeric character. `\W` Matches any non-alphanumeric.
- `\s`: Matches any whitespace character. `\S` Matches any non-whitespace character.
- `\b`: Matches a word boundary.
- `|`: OR. For example, `cat|dog` matches 'cat' or 'dog'.
- `()`: Groups characters together. For example, `(ab)+` matches 'ab', 'abab', 'abababb', etc.

For instance, `h.*o` would match "hello", "hllo", "haallo", etc. because `.*` matches zero or more of any character between 'h' and 'o'. In another example `^\b\w+\b$` would match only lines that contain a single word and nothing else.

Info

Escaping metacharacters
What if you want to match a metacharacter literally? For instance, you want to match a literal dot, `.`. In this case, you need to escape the metacharacter with a backslash, like `\.`. This tells the regex engine to treat the dot as a literal character, not as a metacharacter.

2.3 Capture Groups

A common usecase of regex is to extract parts of a string that match a pattern. To do this, we use capture groups. A capture group is a part of the pattern enclosed in parentheses. When a pattern with capture groups is matched, the matched text within the parentheses is stored in a group.

The `re.search` function returns a match object, which has a `groups()` method that returns a tuple of the captured groups.

The `re.sub` function can also use capture groups to replace text. The replacement string can reference the captured groups using `\1`, `\2`, etc. to refer to the first, second, etc. captured group. The `\0` reference the entire match.

Example

```
import re
text = "The dogs and the cats"
```

```python
# Lets extract the first two 4-letter words
# We will also capture the text between the words
pattern = r"(\b\w{4}\b)(.*)(\b\w{4}\b)"
matches = re.search(pattern, text)
print(matches.groups())
# Now lets swap the words
new_text = re.sub(pattern, r"\3\2\1", text)
print(new_text)
```

Output

```
('dogs', ' and the ', 'cats')
The cats and the dogs
```

Advanced

You can use captured groups in the pattern itself
For instance, the pattern `^(\b\w+\b).*\1$` matches lines that start and end with the same word. The `\1` reference the first captured group, which is the first word in the line.

```python
import re
text = "The quick brown fox jumps over the lazy dog."
pattern = r"^(\b\w+\b).*\1$"
print(re.match(pattern, text) is not None)
text = "people often understimate other people"
print(re.match(pattern, text) is not None)
```

Output

```
False
True
```

3 Error Handling

Error handling is the process of responding to errors that occur during program execution.

Advice

When error handling is neglected
It is easy to overlook this aspect of programming when you are a beginner, it is one of those things that sound like an unnecessary nuisance. Which in part explains the sad state of software in our current day to day. Error handling makes your code robust, reliable, maintainable and (surprisingly, in the case of Python) more readable. Bad error handling has cost billions of dollars, crashed planes, killed people, tanked stock markets and downed satellites. This website is dedicated exclusively to list horror stories of bad error handling in software, each story has the monetary cost of the error as title. If you like videos as a learning resource, I suggest you watch this one: 25 crazy software bugs explained. With some luck, those will scare you straight :P

3.1 Exceptions

Exception handling allows a program to deal with unexpected situations and errors gracefully. It separates error-handling code from regular code. For instance, as part of their execution many algorithms will try to allocate some memory, and if that fails, they will raise an exception. Or perhaps a script is trying to get some data from the internet, but the provided URL is not reachable. We do not have access to the implementation of these algorithms, nor do we care about it. Luckily, the exception that will be raised gives us the information we need and a chance to handle the error.

For instance, if a network resource cannot be accessed, the program can inform the user, wait a few seconds and try again. If the program cannot recover from the error, it can log the error and exit gracefully.

3.1.1 Basic keywords: `try`, `except`, `raise`

Python uses three primary keywords for exception handling:

- `try`: Defines a block of code to monitor for exceptions.
- `except`: Catches exceptions *raised* (that occur in) the `try` block.
- `raise`: Raises an exception manually.

On top of these, there are two optional keywords:

- `else`: Executes if no exceptions are raised.
- `finally`: Always executes, regardless of whether an exception was raised.

Example

```python
def divide(a, b):
    try:
        result = a / b
    # We can catch specific exceptions
    except ZeroDivisionError:
        print("Error: Division by zero")
        return None
    # And/or catch "any" exceptions
    except Exception as e:
        # By adding "as e" we can access the exception object
        print(f"An unknown error occurred: {e}")
        return None
    else:
        print(f"Result: {result}")
        return result
    finally:
        print("This always runs")

divide(10, 2)
divide(10, 0)
```

Output

```
Result: 5.0
This always runs
Error: Division by zero
This always runs
```

Advice

99% of the time you will just want to use the `try` and `except` blocks.
The `else` block is useful when you want to execute some code only if no exceptions were raised.
The `finally` block is useful for cleanup code that should always run, regardless of whether an exception was raised. For instance, closing a file or a database connection.

Exceptions, like `ZeroDivisionError` above, are classes that inherit from the base `Exception` class. One can also raise custom exceptions by creating a new class that inherits from `Exception`, but we will not know how to do so until we learn about Object Oriented Programming.

Functions (and operators are just functions in disguise) can raise exceptions when they encounter an error. Their documentation will tell you what exceptions they can raise. For instance, the `open` function raises a `IOError` if the file does not exist or cannot be opened.

Info

Common Built-in Exceptions
Python provides countless built-in exceptions for different error types, some of the most common are:

- `ValueError`: Raised when an operation receives an argument with the right type but inappropriate value
- `TypeError`: Raised when an operation is performed on an object of inappropriate type
- `FileNotFoundError`: Raised when a file or directory cannot be found
- `IndexError`: Raised when trying to access an invalid index in a sequence
- `KeyError`: Raised when a dictionary key is not found
- `ZeroDivisionError`: Raised when dividing by zero
- `AttributeError`: Raised when an attribute reference or assignment fails

Examples:

We can catch specific exceptions to handle them differently:

```
try:
    with open("nonexistent_file.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
      print("File not found")
```

Output

```
File not found
```

Or we can raise exceptions manually:

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age
try:
    validate_age(-5)
except:
    print("Invalid age")
```

Output

```
Invalid age
```

Note that the `except` block without any arguments catches any exception.

Advice

Best Practices for Exception Handling

1. Keep try blocks small. Only wrap code that might raise an exception.
2. Don't hide errors silently. Log errors or inform users when something goes wrong.
3. Use specific exceptions when possible. This makes it easier to handle different error cases.
4. It is considered a bad practice to catch all exceptions with a generic `except` block. Some exceptions are not meant to be caught (such as the ones coming from assertions).

If you find yourself writing a lot of "try..catch" blocks, this is usually a smell of bad design. The best way to go about error handling is to "design errors out of existence".

Advanced

The most diabolical antipattern

Check out this code:

```
try:
    some_function()
except:
    pass
```

This code will catch all exceptions and simply ignore them. Perhaps the programmer knew that the function could raise an exception, but decided that in such a case the consequences were not important. However, by writing this code, the programmer is hiding ALL errors that might occur at this time.

Instead, you can capture specific exceptions and handle them accordingly. Or, at the very least, log the error by printing some detailed message before moving on.

You can read more about this here.

Advanced

Creating Custom Exceptions

You can create your own exception classes for specific error cases:

```
class InvalidAgeError(Exception):
    """Raised when age is invalid"""
    pass

def validate_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative")
    if age > 150:
        raise InvalidAgeError("Age seems unrealistic")
    return True
```

Warning

Exceptions are designed exclusively to be used to report failure to complete a given task that the code in question is not able to handle on its own. Never use exceptions for normal control flow or to return values.

3.2 Assertions: The `assert` keyword

Assertions are used to test assumptions within the code. If an assertion fails, an `AssertionError` is raised. They are primarily for debugging purposes – checks made during development that should always be true.

Example

```
# This is ok
assert len([1, 2, 3]) == 3
assert len("hello") == 5
# This will raise an AssertionError and halt the program
assert 1 == 2, "This should always fail"
```

Advice

Exceptions vs. Assertions

- Use exceptions for handling runtime errors that can occur in normal program execution.
- Use assertions to check for conditions that should always be true. Assertions are not meant to be caught or handled; they indicate a bug in the code.
- Assertions are also an interesting way to document your code. They are a way to say "this should always be true".

Example

```
def calculate_final_grades(grades):
    assert len(grades) > 0, "Grades cannot be empty"
    assert all(0 <= g <= 100 for g in grades), "Grades must be between 0 and 100"
    return sum(grades) / len(grades)
```

See, if this function was called with an empty list or with a grade outside the 0-100 range, it would be a bug in the calling code. The assertion also serves to indicate to the reader about the restrictions of the function.

4 Exercises

4.1 Regex fun

Goal

Let us practice regex with a few exercises.

Milestone

Write a regular expression that matches the green parts of the following strings:

Regex example

Test strings:

The cat is black
There is a catch
There is nothing here
1234cat4321cat

Write a python function that takes a string and a pattern and returns `True` if the pattern is found in the string, `False` otherwise.

hint

- Copy the test strings to the regex tester at regex101.com and try to write a pattern that matches all the cases. Remember to select Python mode.
- Use the `re.search` function to search for the pattern in the string.

Milestone

Write a regular expression that matches the green parts of the following strings:

Regex example

Test strings:

The quick brown fox jumps over the lazy dog.
The cat is back
There is nothing here

Write a python function that takes a list of strings and a pattern and returns a list of all matches.

hint

- Use the `re.findall` function to find all matches.

Milestone

Write a function that extracts the green parts of the following strings into a list of floats:

Regex example

Test strings:

---

140 mph (225 km/h)
1.24 mph (2 km/h)
20 mph (32 km/h)
233.00 mph (375.17 km/h)

Use the `re.search` function to extract the number from the string. Remember that the `group` method of the match object returns the matched string.

hint

- Use capture groups to extract the number.
- The `*` metacharacter matches **zero or more** occurrences of the preceding character.
- The `+` metacharacter matches **one or more** occurrences of the preceding character.

Milestone

Redo the previous milestone, but this time extract the numbers for all cases in a single line, using `re.findall`.

Milestone

Replace the green parts of the following strings with "dog":

Regex example

Test strings:

The quick brown fox jumps over the lazy dog.
The cat is back
There is nothing here

hint

Use the `re.sub` function to replace the matched strings.

Advanced milestone

The prime finder
The following function uses a regular expression to check if a number is prime:

```
import re
def is_prime_re(n):
  return not re.match(r"^.?$|^(..+?)\1+$", "1" * n)
# Tests
assert(is_prime_re(31))
assert(not is_prime_re(512))
```

Can you explain how this function works? What does the regex pattern `^.?$|^(..+?)\1+$` do?

Warning

This is a really challenging exercise. By no means is this a common use of regex, and I do not expect you to be able to pull this off. It is also a really bad idea to use regex for this task, as it is not efficient. This is just for fun :P.

hint

- The construct `"1" * n` creates a string of `n` ones, but the regex will work for any string, as long as it has `n` identical characters.
- The `|` operator in regex means "or", the left side is checking for a special case.

4.2 Error handling

Goal

Let us practice error handling with a few exercises.

Milestone

Imagine that this function, that allows to set a theme (whatever that entails), is part of some larger application.

```
def set_theme(theme="default"):
    valid_themes = ["default", "light", "dark", "blue"]
    if theme not in valid_themes:
        raise ValueError(f"Invalid theme: {theme}")
    print(f"Theme set to {theme}")
```

Write a script that has a command-line option, `--theme`, that calls the function with the provided theme. If the theme is invalid, catch the exception and call `set_theme()` before the program exits.

hint

- Use the `argparse` module to parse command-line arguments.
- Use a `try` block to call the function and a `except` block to catch the exception.

Milestone

You received the following data as part of the response of an API call:

```
data = [10, "20", "not available", 30, "40.5", None, "0.1", "error"]
results = process_numbers(data)
```

The API promised to send you back a list of numbers, but (as you can see) it sent you a list of strings, where some of them are not numbers. Write a function (`process_numbers`) that processes the data and returns a list of floats. If a string cannot be converted to a float, catch the exception and print an error message, otherwise ignoring the invalid entries.

hint

- Use a `try` block to convert the string to a float and a `except` block to catch the exception.
- Use the `float` function to convert a string to a float.