Lecture VI - Using Modules and Packages

Programming with Python

Dr. Tobias Vlćek
Kühne Logistics University Hamburg - Fall 2025

Quick Recap of the last Lecture

Exceptions and Error Handling

- Exceptions are discovered errors during program execution
- Common built-in exceptions: `ValueError`, `TypeError`, etc.

...

```
x = int("Hello, World!")
```

...

>ValueError: invalid literal for int() with base 10: 'Hello, World!'

Try-Except Blocks

- `try-except` blocks are used to handle exceptions
- `try` block contains code that might raise an exception
- `except` block contains code executed if an exception occurs

...

```
try:
    # Code that might raise an exception
    # ...
except ExceptionType as e:
    # Code to handle the exception
    # ...
except Exception as e:
    # Code to handle any other exceptions
    # ...
```

Raising Exceptions

- We can raise exceptions using the `raise` statement
- Allows for more controlled error handling
- Can include custom error messages

...

```
raise ValueError("This is a custom error message")
```

...

> **Note**
>
> The type if raised exception has to exist or you have to create a custom error type before.

Assertions

- Assertions check if a condition is true
- If the condition is false, an `AssertionError` is raised
- Useful for checking calculations or variable types

...

```
x = -1
assert x > 0, "x must be positive"
```

...

Question: Will this raise an AssertionError?

Debugging

- Debugging is the process of finding and fixing errors in code
- Using `print` and `assert` statements
- Using logging
- Using built-in debugging tools in IDEs

...

> **Tip**
>
> That's why IDEs are so helpful in coding.

Modules

Why Modules?

- Modular programming breaks large tasks into smaller subtasks
- Modules are like building blocks for larger applications
- Individual modules can be combined to create a complete program
- This approach enhances code organization and reusability

Creating Modules

- Modules are simply `.py` files containing Python code
- They can define functions, classes, and variables
- They can be imported into other Python scripts

```python
# The script new_module.py is in the same directory as this script
import lec_06_new_module as new_module # Here we import the module
new_module.my_function() # Here we call the function from the module
```

```
Hello from my_function!
```

Importing functions from modules

- We can also import specific functions from a module
- This is useful if we only need a few functions from a module
- Analogously, we can import classes or variables from a module

...

```python
# Multiple imports from a module are possible as well!
from lec_06_new_module import another_function, yet_another_function
another_function()
yet_another_function()
```

```
Hello from another_function!
Hello from yet_another_function!
```

...

> **Tip**
>
> This is a good way to avoid importing too much from a module. In addition, we don't need to use the module name before the function name when we use the functions from the module.

Built-in Modules

Python comes with many built-in modules. Common ones include:

Module	Description
`math`	Different mathematical functions
`random`	Random number generation
`datetime`	Date and time manipulation
`os`	Operating system interaction
`csv`	Reading and writing CSV files
`re`	Regular expression operations

Importing from the Standard Library

Task: Use Python's `math` module to calculate the area of a circle.

```
# Import the `math` module.
# Define a function named `calculate_area` that takes the radius `r` as an
argument.
# Inside the function, use the `math.pi` constant to get the value of π.
# Calculate the area in the function and return it.

# Your code here

assert calculate_area(5) == 78.53981633974483
```

...

> Tip
>
> Note, how assertations can be used to check if a function works correctly.

Standard Libraries

Random Numbers

The `random` module provides functions for random numbers

- `random.random()`: random float between 0 and 1
- `random.uniform(a, b)`: random float between `a` and `b`
- `random.randint(a, b)`: random integer between `a` and `b`
- `random.choice(list)`: random element from a list
- `random.shuffle(list)`: shuffle a list

> Tip
>
> There are many more functions in the `random` module. Use the `help()` function to get more information about a module or function.

Random Numbers in Action

Task: Time for a task! Import the `random` module and create a small number guessing game with the following requirements:

```
# TODO: Implement a random number guessing game.
# The game should work as follows:
# - The computer selects a random number between 1 and 10
# - The user has to guess the number and has three guesses
# - The computer tells the user whether their guess is too high, too low,
or correct
# - The computer should also print how many guesses the user made before
guessing the number correctly
# - It should also ask the user if they want to play again
# Your code here
```

...

> Tip
>
> Remember, that the input function always returns a string!

OS Module

- The `os` module provides functions to interact with the OS
- `os.listdir(path)`: list all files and directories in a directory
- `os.path.isfile(path)`: check if a path is a file
- `os.path.exists(path)`: check if a path exists
- `os.makedirs(path)`: create a directory

...

> Tip
>
> These can be quite useful for file handling. The `os` module contains many more functions, e.g. for changing the current working directory, for renaming and moving files, etc.

CSV Module

- Comma-Separated Values files are used to store tabular data
- Write: `csv.writer(file)`
- Read: `csv.reader(file)`

...

```
import csv # Import the csv module

with open('secret_message.csv', 'w') as file: # Open the file in write mode
    writer = csv.writer(file) # Create a writer object
    writer.writerow(['Entry', 'Message']) # Write the header
    writer.writerow(['1', 'Do not open the file']) # Write the first row
    writer.writerow(['2', 'This is a secret message']) # Write the second
row
```

...

Task: Copy the code and run it. Do you have a new file?

OS and CSV Module in Action

Task: Time for another task! Do the following:

```
# First, check if a directory called `module_directory` exists.
# If it does not, create it.
# Then, list all files in the current directory and save them in a CSV file
called `current_files.csv` in the new `module_directory`.
```

```
import os
if not os.path.exists('module_directory'):
    pass
# Your code here
```

Regular Expressions

Why Regular Expressions?

Let's see the limitations of basic string methods:

```
text = "Contact us at: tobias@beyondsimulations.com or call 123-456-7890"

# Find email - how would you do this with basic string methods?
# Find phone number - what about this?
# What if there are multiple emails with different formats?
```

...

Regular expressions solve these problems by finding patterns! Let's work with this sample text throughout our examples:

...

```
sample_text = """
User john123 logged in at 2024-01-15
User mary_doe logged in at 2024-01-16
User bob logged in at 2024-01-17
Error: user invalid_user! failed login
"""
```

Using Regular Expressions

- `re.search(pat, str)`: search for a pattern in a string
- `re.findall(pat, str)`: find all occurrences of a pattern
- `re.fullmatch(pat, str)`: check if entire string matches pattern
- `re.sub(pat, repl, str)`: replace a pattern in a string
- `re.split(pat, str)`: split a string by a pattern

...

> **Note**
>
> As always, there is more. But these are a good foundation to build upon.

Literal Matching

```
import re

sample_text = """
```

```
User john123 logged in at 2024-01-15
User mary_doe logged in at 2024-01-16
User bob logged in at 2024-01-17
Error: user invalid_user! failed login
"""

# Find exact word "User"
result = re.findall(r'User', sample_text)
print(f"Found 'User': {result}")
```

```
Found 'User': ['User', 'User', 'User']
```

...

Task: Find all occurrences of “logged” in the sample text.

Special Character: The Dot (.)

The `.` matches any single character:

```
sample_text = """
User john123 logged in at 2024-01-15
User mary_doe logged in at 2024-01-16
User bob logged in at 2024-01-17
Error: user invalid_user! failed login
"""

# Find "User" followed by any character, then "o"
result = re.findall(r'bo.', sample_text)
print(f"Found 'bo.': {result}")  # This finds "bob"
```

```
Found 'bo.': ['bob']
```

...

Task: Use `.` to find all 4-letter words starting with “use”.

...

> Tip
>
> The dot is like a wildcard - it fills in for any single character you don’t know.

Character Classes: `[abc]`

Square brackets match any character inside them:

```
# Find usernames that start with 'j' or 'm'
result = re.findall(r'User [jm]\w+', sample_text)
print(f"Users starting with j or m: {result}")
```

```
Users starting with j or m: ['User john123', 'User mary_doe']
```

...

Useful character classes:

- `[abc]` - matches a, b, or c
- `[a-z]` - matches any lowercase letter
- `[0-9]` - matches any digit
- `[a-zA-Z0-9]` - matches letters and numbers

...

Task: Find all years.

Common Pattern Shortcuts

Instead of writing `[0-9]`, we can use shortcuts:

Shortcut	Meaning	Same as
`\d`	Any digit	`[0-9]`
`\w`	Word character	`[a-zA-Z0-9_]`
`\s`	Whitespace	`[ \t\n]`

...

```python
# Find dates using \d (much cleaner!)
result = re.findall(r'\d\d\d\d-\d\d-\d\d', sample_text)
print(f"Found dates: {result}")
```

```
Found dates: ['2024-01-15', '2024-01-16', '2024-01-17']
```

...

Task: Find all usernames using `\w`.

Quantifiers: How Many Times?

Instead of repeating `\d\d\d\d`, we can specify quantities:

Quantifier	Meaning
`{4}`	Exactly 4 times
`{2,4}`	Between 2 and 4 times
`+`	One or more times
`*`	Zero or more times
`?`	Zero or one time

...

```
# Much cleaner date pattern!
result = re.findall(r'\d{4}-\d{2}-\d{2}', sample_text)
print(f"Dates with quantifiers: {result}")
```

```
Dates with quantifiers: ['2024-01-15', '2024-01-16', '2024-01-17']
```

...

Task: Find usernames of any length with `\w+`.

Putting It Together: Email Finder

Let's build an email pattern step by step!

...

```
email_text = "Contact: john@email.com, mary.doe@company.org, or
bob@test.co.uk"

# Step 1: Basic pattern
basic = r'\w+@\w+\.\w+'
print("Basic emails:", re.findall(basic, email_text))
```

```
Basic emails: ['john@email.com', 'doe@company.org', 'bob@test.co']
```

...

```
# Step 2: Handle dots in names
better = r'[\w.]+@\w+\.\w+'
print("Better emails:", re.findall(better, email_text))
```

```
Better emails: ['john@email.com', 'mary.doe@company.org', 'bob@test.co']
```

...

```
# Step 3: Handle multiple domain parts
best = r'[\w.]+@[\w.]+\.\w+'
print("Best emails:", re.findall(best, email_text))
```

```
Best emails: ['john@email.com', 'mary.doe@company.org', 'bob@test.co.uk']
```

Debugging Regular Expressions

When your regex doesn't work:

1. Test with simple examples first
2. Build the pattern gradually
3. Use online tools like regexr.com
4. Print intermediate results

...

> Tip
>
> Remember: regex can be complex, but you don't need to master everything at once. Start simple and build up!

Regex in Action

Task: Replace all occurences of `Python` by "SECRET".

```
import re
string = """
Python is a programming language.
Python is also a snake.
Monty Python was a theater group.
"""
# Your code here
```

Advanced Regex in Action

Task: Use regular expressions to extract all dates from the text.

```
dates = """
On 07-04-1776, the United States declared its independence. Many years
later,
on 11-09-1989, the Berlin Wall fell. In more recent history, the COVID-19
pandemic was declared a global emergency on 04-11-2020.
"""
# Try to find all dates in the above text with findall()
# Your code here
```

Packages

What are Packages?

- Packages are esentially collections of modules
- They can contain multiple modules, subpackages, and data files
- Many packages are available in the Python Package Index (PyPI)
- You don't have to invent the wheel yourself
- A lot of functionality is already implemented by others!

Installing Packages

- Packages are installed in the shell
- Use `uv add <package_name>` to install a specific package
- Afterward you can import from the package in your scripts

...

Task: Install the `pandas` and `numpy` packages, which are commonly used for data analysis. We will use them together next week!

...

```{bash}
uv add pandas numpy
```

...

> Tip
>
> If you install packages like this, you can use the shell to do so! Alternatively, you can use `uv add <package_name>` in the terminal in the IDE.

Virtual Environments

- Virtual environments isolate a project's dependencies
- With uv, the environment is created and managed automatically
- No manual venv creation needed and others can replicate
- Especially important when working on several projects at once

...

Common uv command you'll use:

```{bash}
# Install dependencies from pyproject.toml (to use the code from others)
uv sync

# Start a Python REPL in the project environment (you can then work
directly here)
uv run python
```

...

> Note
>
> And that's it for today's lecture! We now have completed the first step into data science in Python. Next week, we can use this new knowledge to start to work with some tabular data and matrices.

Literature

Interesting Books

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. Link to free online version
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

...

> Tip
>
> Nothing new here, but these are still great books!

. . .

For more interesting literature to learn more about Python, take a look at the literature list of this course.