PROGRAMMING IN PYTHON I

Recursion, Generators and Modules

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: schoergenhumer@ml.jku.at
Write mails only for personal questions
Institute ML Homepage

Recursion

- A **recursive function** is a function that **calls itself** (either directly or indirectly, e.g., via other function calls)
- Example: power function x^y (for simplicity, assume y is an integer and at least 1)
- Idea: Split the task into a smaller **subtask** and a **rest**: $x^y = x * x^{y-1}$
- We know how to multiply, so we can solve this subtask immediately and we assume that the rest (x^{y-1}) is taken care of by our function.
- We let our function repeat this process, i.e., splitting into subtask and rest until we reach a state where the rest can also be solved immediately: x^y with $y = 1$ is $x^1 = x$. This is called a **base case** or **recursion anchor**.

Recursive Power Function

```
def power(x, y):
    if y == 1:  # base case
        return x
    return x * power(x, y - 1)  # recursive call
```

- Example call with `power(2, 4)`:

```
2 * power(2, 3)
    -----------
    2 * power(2, 2)
        -----------
        2 * power(2, 1)
            -----------
            2
```

- End of recursion reached: evaluate “backwards” until the initial `power(2, 4)` call: 2*2*2*2 = 16
- Always implement the end of a recursion, otherwise, this will result in an endless recursion

Recursive Functions

- There can be **multiple recursive calls** in the same function, either in different branches or even subsequent calls
- There can be **multiple ends** of a recursion (multiple base cases)
- Especially useful if the problem is already defined in a “recursive” way, e.g.:
  - Traversing through a tree-like data structure
  - Processing (potentially arbitrarily) nested data structures
  - Problems that can be solved with the “divide and conquer” principle (e.g., merge sort)

Generator Functions (1)

- Writing **yield** within a function instead of **return** will make the function a so-called **generator function** that returns a generator iterator object

```
def generate_str_numbers(n):
    for i in range(n):
        yield str(i)
```

- The code of a generator function is actually only executed when an element is requested, e.g., via a for loop:

```
gen = generate_str_numbers(3)  # No execution
for str_number in gen:  # Code is executed
    # Do something
```

- After reaching a **yield**, the specified value is returned and the **execution is suspended** until the **next element** is requested and the function is **resumed again**

Generator Functions (2)

- Generator functions yield elements when **needed**, rather than processing everything at once and returning the entire result as, e.g., a list (can save memory)
- This allows for infinite generators:

```
def infinite():
    import random
    while True:
        yield random.random()
```

- Elements of generators can also be accessed using the built-in function `next(...)`:

```
my_rnd_generator = infinite()
for _ in range(3):
    rnd = next(my_rnd_generator)
    print(rnd)
```

- If there are no more elements and you still call `next(...)`, this will result in an exception (`StopIteration`)

Modules

- Often, we want to reuse code (e.g., a function) in different programs and projects
- In Python, we can do this by putting the function into a separate file (**module**)
- Naming convention for modules: lowercase letters + underscores (if needed)
- We can then load (**import**) this function definition from the file into our code file
- There are many modules with lots of functionalities available
  - You will write your own modules
  - We will learn about some important modules

Example for Importing a Module (1)

- Consider the file `my_module.py` in the same directory where your code is[1]
- Assume that the file contains a function `add`
- There are now several ways to import from this module:

```
import my_module  # Can use everything within
my_module.add(...)

import my_module as mm  # Same but renamed
mm.add(...)

from my_module import add  # Can only use add
add(...)  # No module specification needed

from my_module import add as my_add
my_add(...)  # Same but renamed
```

[1] Python also searches for modules in other places

Example for Importing a Module (2)

- Python files can also be put into **packages** for better structuring your project
- Naming convention for packages: lowercase letters (underscores are discouraged)
- Packages are directories/folders that contain an empty `__init__.py` file[2]
- Consider the same file `my_module.py` but in a package called `mypackage`
- Again, there are several ways to import from this module (see next slide)

[2] They need not be empty, e.g., some initialization

Example for Importing a Module (3)

```
import mypackage.my_module
mypackage.my_module.add(...)

import mypackage.my_module as mm
mm.add(...)

from mypackage import my_module
my_module.add(...)

from mypackage import my_module as mm
mm.add(...)

from mypackage.my_module import add
add(...)

from mypackage.my_module import add as my_add
my_add(...)
```

Code Execution upon Importing

- When importing a module, all code within is executed
- If you want this code only to be executed when running the module as a script (as your main starting code, e.g., via `python my_module.py`) and not when you import the module, you have to create this conditional check:

```
if __name__ == "__main__":
    # Code that is only executed when run as a
    # script. It is not run when being imported
```