Astroinformatics II (Semester 2 2024)

Writing Efficient Code in Python

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

September 10, 2024

Motivation

➕ benefits of Python:

- high-level, interpreted programming language
- versatile
- many libraries available

➖ drawback of Python:

slow(er than many compiled languages)

Motivation

+ benefits of Python:

- high-level, interpreted programming language
- versatile
- many libraries available

− drawback of Python:

slow(er than many compiled languages)

Writing efficient Python code is crucial to **maximize the performance** of your programs and make the best use of resources.

What is Efficient Code?

Efficient refers to code that satisfies two key concepts:

fast code: low latency between starting and returning the result

resourceful code: making best use of memory, cores

What is Efficient Code?

Efficient refers to code that satisfies two key concepts:

fast code: low latency between starting and returning the result

resourceful code: making best use of memory, cores

⇨ reducing both latency and overhead

What is Efficient Code?

Efficient refers to code that satisfies two key concepts:

fast code: low latency between starting and returning the result

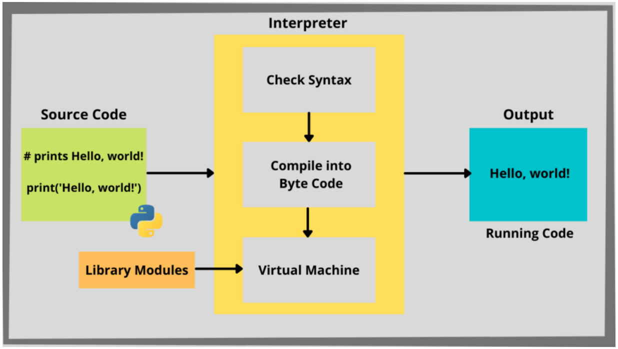
resourceful code: making best use of memory, cores

⇒ reducing both latency and overhead

Additionally: writing code that is well readable by following the best practices and guiding principles of Python

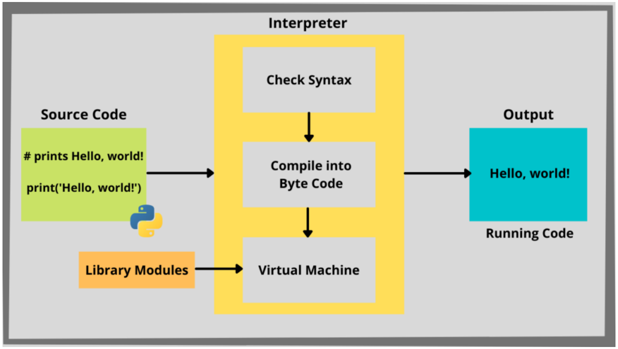
⇒ **Pythonic** code: code that follows the conventions and idioms of the Python community, resulting in code that is clear and efficient.

Understanding Python's Execution Model



Parsing
The Python interpreter first reads the Python code. Syntax checks are carried out. The code is parsed into an Abstract Syntax Tree (AST), representing the code structure in a tree-like form, making it easier for the compiler to understand.

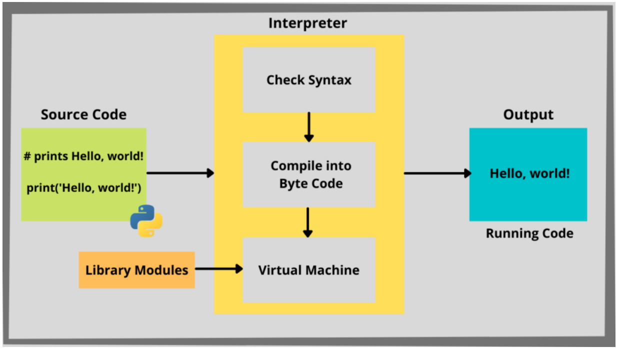
Understanding Python's Execution Model



Compilation
The AST is compiled into bytecode, a low-level, platform-independent representation of your code in a fixed set of instructions that represent arithmetic, comparison, memory operations, etc. The byte code instructions are created in the `.pyc` file which Python handles internally. It can be viewed with the following command:

```
python -m py_compile mycode.py
```

Understanding Python's Execution Model



Interpreter - the Python Virtual Machine (PVM)
The Python virtual machine is the runtime engine of CPython. It executes the bytecode produced during the compilation stage. The PVM is an interpreter for the bytecode, going through the instructions one by one and performing the specified operations.

Understanding Python's Execution Model

Why exactly is Python slow?

Understanding Python's Execution Model

Why exactly is Python slow?

- Python is dynamically-typed and interpreted. The virtual machine incurs overhead, because it has to be prepared for the datatypes to potentially change. It also has to wrap up low-level types with higher-level functions like hashing and printing. This is why compilation can help for code that has lots of loops making lots of calls.

  In order to take advantage of compilation, you need to be more careful with specifying your datatypes (ints, floats, strings), so that you can remove the need for the flexibility of the interpreter.

Understanding Python's Execution Model

Why exactly is Python slow?

- Python is dynamically-typed and interpreted. The virtual machine incurs overhead, because it has to be prepared for the datatypes to potentially change. It also has to wrap up low-level types with higher-level functions like hashing and printing. This is why compilation can help for code that has lots of loops making lots of calls.

  In order to take advantage of compilation, you need to be more careful with specifying your datatypes (ints, floats, strings), so that you can remove the need for the flexibility of the interpreter.
- The efficiency and performance can be affected by factors such as the complexity of the Python code, the use of built-in functions (which are typically optimized C functions), and the interaction with external modules and libraries.

Writing Efficient Python Code

Choosing the Right Data Structures and Algorithms

Python offers a variety of data structures. The choice of data structures and algorithms has a significant impact on the efficiency of your code.
For example, lists are great for small collections, but for large datasets with non-sequential access, a set or dictionary might be more efficient. Tuple are immuatable, and more light weighted than list, if the when order of objects won't change, tuple is preferred.

Choosing the right algorithm can significantly improve the performance of your program.
For example, if the data is ordered, binary search would reduce the time complexity to $\mathcal{O}(\log N)$ compare linear search $\mathcal{O}(N)$. Recursive functions are often more readable than iteration, but the latter is usually more efficient because it doesn't involve the overhead of pushing and popping frames on the call stack.

Measure Code Performance

For optimizing code, it is important to know how to **measure** code performance.

In the following, we will see methods on how to measure and compare runtimes between different coding approaches, as well as measure memory consumption.

After that, we will see how to replace these bottlenecks with more efficient Python code.

Measure Code Execution Time

By analyzing code runtimes, we can find bottlenecks and can be sure to implement code that is fastest and thus most efficient.

To compare runtimes, we need to be able to compute the runtime for a line or multiple lines of code. **IPython** comes with some handy built-in so-called **magic commands** we can use to time our code. Magic commands are enhancements that have been added on top of the normal Python syntax. They are prefixed with the percentage sign.

Measure Code Execution Time

example: We want to inspect the runtime for selecting 1000 random numbers between zero and one using NumPy's `random.rand()` function. Using the **magic command** `%timeit` just requires adding it before the line of code we want to analyze.

```
%timeit rand_nums = np.random.rand(1000)
```

```
10.6 µs ± 103 ns per loop (mean ± std. dev. of 7 runs,
100,000 loops each)
```

Measure Code Execution Time

example: We want to inspect the runtime for selecting 1000 random numbers between zero and one using NumPy's `random.rand()` function. Using the **magic command** `%timeit` just requires adding it before the line of code we want to analyze.

```
%timeit rand_nums = np.random.rand(1000)
```

```
10.6 µs ± 103 ns per loop (mean ± std. dev. of 7 runs,
100,000 loops each)
```

One of the advantages of `%timeit` is that it provides an average of timing statistics over multiple runs and loops were generated.
This provides a more accurate representation of the actual runtime rather than relying on just one iteration to calculate the runtime.

Measure Code Execution Time

Specifying number of runs/loops

The number of runs represents how many iterations you want to estimate the runtime. The number of loops represents how many times you want the code to be executed per run.
We can specify the number of runs, using the `-r` flag, and the number of loops, using the `-n` flag.
In this example, `%timeit` would execute our random number selection 20 times in order to estimate runtime (2 runs each with 10 executions).

```
# Set number of runs to 2 (-r2)
# Set number of loops to 10 (-n10)
%timeit -r2 -n10 rand_nums = np.random.rand(1000)
```

```
54.8 µs ± 15.9 µ s per loop (mean ± std. dev. of 2 runs,
10 loops each)
```

Measure Code Execution Time

We can also run `%timeit` over multiple lines of code by using two percentage signs:

```
%%timeit
# Multiple lines of code
nums = []
for x in range(10):
    nums.append(x)
```

```
1.44 µs ± 19 ns per loop (mean ± std. dev. of 7 runs,
1,000,000 loops each)
```

Measure Code Execution Time

With the magic command `%timeit`, we can see the total time used for a sngle line or a block of code. But, what if we wanted to time a large code base or see the line-by-line runtimes within a function? For this, we use a concept called **code profiling** that allows us to analyze code more efficiently.

Code profiling is a technique used to describe how long, and how often, various parts of a program are executed. We'll focus on the `line_profiler` package to profile a function's runtime line-by-line.

Measure Code Execution Time

With the magic command `%timeit`, we can see the total time used for a sngle line or a block of code. But, what if we wanted to time a large code base or see the line-by-line runtimes within a function? For this, we use a concept called **code profiling** that allows us to analyze code more efficiently.

Code profiling is a technique used to describe how long, and how often, various parts of a program are executed. We'll focus on the `line_profiler` package to profile a function's runtime line-by-line.

Since this package isn't a part of Python's Standard Library, we need to install it first.

Measure Code Execution Time

Let's explore using line_profiler with an example. Suppose we have a list of names along with each someones height (in centimeters) and weight (in kilograms) loaded as NumPy arrays.

```
names = ['Charles', 'Pedro', 'Dan']
hts = np.array([188.0, 191.0, 185.0])
wts = np.array([ 95.0, 100.0, 75.0])
```

We create a function convert_units that converts each person's height from centimeters to inches and weight from kilograms to pounds.

```
def convert_units(names, heights, weights):
    new_hts = [ht * 0.39370 for ht in heights]
    new_wts = [wt * 2.20462 for wt in weights]
    data = {}
    for i,name in enumerate(names):
        data[name] = (new_hts[i], new_wts[i])
    return data
```

Measure Code Execution Time

We then can profile our function with the `line_profiler` package.

The following **example** shows the usage:

```
%load_ext line_profiler
%lprun -f convert_units convert_units(names, hts, wts)
```

To use this package, we first need to load it into our session. We can do this using the command `%load_ext` followed by `line_profiler`.

Now, we can use the magic command `%lprun` to gather runtimes for individual lines of code within the `convert_units` function.
`%lprun` uses a special syntax. First, we use the `-f` flag to indicate we'd like to profile a function. Then, we specify the name of the function we'd like to profile. Note, the name of the function is passed without any parentheses.

Finally, we provide the exact function call we'd like to profile by including any arguments that are needed.

Memory Management in Python

Before we use profiling for tracking memory usage, let's first understand the fundamentals of **Memory Management in Python**.

Memory Management in Python

Before we use profiling for tracking memory usage, let's first understand the fundamentals of **Memory Management in Python**.

The **private heap** is at the core of Python's memory management. It is where all Python objects and data structures are stored. Programmers cannot access this private heap directly; instead, they interact with objects through Python's Memory Management System. It uses:

- **Memory allocators:** Python uses a built-in memory allocator that manages the allocation and deallocation of memory blocks. This allocator optimizes for small objects using free lists, which recycle previously allocated memory blocks to speed up future allocations. For more complex objects, such as lists and dictionaries, Python employs dynamic memory allocation to manage their size variations.
- **Memory pools:** Python organizes memory into pools according to object size to minimize overhead and fragmentation. This helps manage memory more efficiently by grouping similar-sized objects.

Memory Management in Python

Python employs a dual approach for **memory deallocation and garbage collection**:

- **Reference counting:** This is the primary method where Python tracks each object's references. When an object's reference count drops to zero, indicating no references to it, the memory allocator immediately frees its memory.
- **Cyclic garbage collector:** Python includes a cyclic garbage collector for managing circular references, which reference counting alone can't handle. This collector periodically identifies and cleans up groups of objects that reference each other but are no longer in use elsewhere in the program.

Measure Memory Usage

Understanding and optimizing memory usage are vital for maintaining application performance.

Python offers several tools for memory profiling, which provide insights beyond standard debugging, helping developers identify and resolve memory inefficiencies:

`pympler`: A comprehensive tool that tracks memory usage and analyzes object space, making it suitable for detailed memory investigations.

`memory_profiler`: This tool offers a line-by-line memory usage analysis, allowing developers to pinpoint exact lines where memory consumption is high.

`tracemalloc`: Integrated into the Python standard library, tracemalloc helps track memory allocations and detect leaks, offering insights into memory allocation trends.

Measure Memory Usage

A basic approach for inspecting memory consumption is using Python's built-in module `sys` which contains system-specific functions, among them `sys.getsizeof()` which returns the size of an object in bytes.

example:

```
import sys
nums_list = [*range(1000)]
sys.getsizeof(nums_list)
```

```
8056
```

```
nums_np = np.array(range(1000))
sys.getsizeof(nums_np)
```

```
8112
```

Measure Memory Usage

However, if we wanted to inspect the line-by-line memory size of our code, we need to use a code profiler.

We'll use the `memory_profiler` package which is very similar to the `line_profiler` package. It comes with a magic command (`%mprun`) that uses the same syntax as `%lprun`.

We first have to install it:

```
#!pip install memory_profiler
```

To be able to apply `%mprun` to a function and calculate the memory allocation, this function should be loaded from a separate physical file and not in the IPython console so first we will create a `utils_funcs.py` file and define the `convert_units` function in it, and then we will load this function from the file and apply `%mprun` to it.

```
%mprun -f convert_units convert_units(names, hts, wts)
```

Pythonic Loops

A `for` loop is used when the number of iterations is predetermined, often iterating over elements using functions like `range`.
In contrast, a `while` loop operates without prior knowledge of iterations and allows for flexible initialization within its body.

```python
import timeit

def whileloop():
    i=0
    while i<10:
        x=1
        i=i+1

def forloop():
    for i in range(0,10):
        x=1

print(timeit.timeit(whileloop))
print(timeit.timeit(forloop))
```

Pythonic Loops

We find that the `for` loop is 2 - 3 times (depending on the Python version) faster than the `while` loop. Why is this the case?

Pythonic Loops

We find that the `for` loop is 2 - 3 times (depending on the Python version) faster than the `while` loop. Why is this the case?

`range()` used in the `for` loop is implemented in C, whereas `i = i+1` is interpreted.

In detail:

In the `while` loop, the comparison `i < 10` is executed in Python, whereas in the `for` loop, the job is passed to the iterator of `range(10)`, which internally does the iteration (and hence bounds check) in C.

In the `while` loop, the loop update `i = i + 1` happens in Python, whereas in the `for` loop again the iterator of `range(10)`, written in C, does the `i+=1 (or ++i)`.

Sequence of Numbers with `range`

The `range()` function generates a sequence of numbers, often used as an iterator in loops. It returns a range object that starts from 0 by default and goes up to (but doesn't include) the specified number.

We can also use this to access elements from a list.

example:

```python
fruits = ["apple", "banana", "cherry"]

for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

Loops with `enumerate()`

An alternative is the `enumerate()` function: It is useful when you want both the index and the value of each element in an iterable.

example:

```python
fruits = ["apple", "banana", "cherry"]

for i, fruit in enumerate(fruits):
    print(f"Index {i}: {fruit}")
```

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

Loops with zip()

Often we have to loop though multiple data structures:

```python
names = ["Alice", "Bob", "Charlie"]
scores = [95, 89, 78]

for i in range(len(names)):
    print(f"{names[i]} scored {scores[i]} points.")
```

Loops with `zip()`

Often we have to loop though multiple data structures:

```python
names = ["Alice", "Bob", "Charlie"]
scores = [95, 89, 78]

for i in range(len(names)):
    print(f"{names[i]} scored {scores[i]} points.")
```

We can use the `zip()` function instead, as in the following **example**:

```python
names = ["Alice", "Bob", "Charlie"]
scores = [95, 89, 78]

for name, score in zip(names, scores):
    print(f"{name} scored {score} points.")
```

The Pythonic version using `zip()` is more elegant and avoids the need for manual indexing, making the code cleaner and more readable.

Comprehensions

In Python, **list comprehensions** provide a concise way to create lists based on existing lists. They can be more readable and faster than traditional loops.

example:
We have a list numbers from which we want to create a squared_numbers list. You can use a for loop:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = []

for num in numbers:
    squared_numbers.append(num ** 2)

print(squared_numbers)
```

```
Output >>> [1, 4, 9, 16, 25]
```

Comprehensions

Using list comprehensions provides a cleaner and simpler syntax for this task. They allow you to create a new list by applying an expression to each item in an iterable.

This **example** gives a concise alternative using a list comprehension expression:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num ** 2 for num in numbers]

print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

Comprehensions

Using list comprehensions provides a cleaner and simpler syntax for this task. They allow you to create a new list by applying an expression to each item in an iterable.

This **example** gives a concise alternative using a list comprehension expression:

```python
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num ** 2 for num in numbers]

print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

List comprehension is in some cases slightly faster, but mostly enhances readability.

List Comprehension with Conditional Filtering

You can also add filtering conditions within the list comprehension expression. In the following **example**, list comprehension creates a new list containing only the odd numbers from the numbers list:

```python
numbers = [1, 2, 3, 4, 5]
odd_numbers = [num for num in numbers if num % 2 != 0]

print(odd_numbers)
```

```
[1, 3, 5]
```

Dictionary Comprehension

With a syntax similar to list comprehension, dictionary comprehension allows you to create dictionaries from existing iterables.

We first see an **example** with a `for` loop:

```python
fruits = ["apple", "banana", "cherry", "date"]
fruit_lengths = {}

for fruit in fruits:
    fruit_lengths[fruit] = len(fruit)

print(fruit_lengths)
```

```
{'apple': 5, 'banana': 6, 'cherry': 6, 'date': 4}
```

Dictionary Comprehension

Let's now write the dictionary comprehension equivalent:

```
fruits = ["apple", "banana", "cherry", "date"]
fruit_lengths = {fruit: len(fruit) for fruit in fruits}

print(fruit_lengths)
```

```
{'apple': 5, 'banana': 6, 'cherry': 6, 'date': 4}
```

This dictionary comprehension creates a dictionary where keys are the fruits and values are the lengths of the fruit names.

Dictionary Comprehension with Conditional Filtering

As with list comprehensions, we can modify our dictionary comprehension expression to include a condition:

```python
fruits = ["apple", "banana", "cherry", "date"]
long_fruit_names = {fruit: len(fruit)
                    for fruit in fruits if len(fruit) > 5}

print(long_fruit_names)
```

```
{'banana': 6, 'cherry': 6}
```

Here, the dictionary comprehension creates a dictionary with fruit names as keys and their lengths as values, but only for fruits with names longer than 5 characters.

Context Managers

Context managers in Python help you manage resources efficiently. With context managers, you can allocate and clean up resources easily. The simplest and the most common example of context managers is in **file handling**.

example:

```
filename = 'somefile.txt'
file = open(filename,'w')
file.write('Something')
```

Context Managers

Context managers in Python help you manage resources efficiently. With context managers, you can allocate and clean up resources easily. The simplest and the most common example of context managers is in **file handling**.

example:

```
filename = 'somefile.txt'
file = open(filename,'w')
file.write('Something')
```

It doesn't close the file descriptor resulting in **resource leakage**.

```
print(file.closed)
```

```
False
```

Context Managers

You'll probably come up with the following:

```
filename = 'somefile.txt'
file = open(filename,'w')
file.write('Something')
file.close()
```

While this attempts to close the descriptor, it does not account for the **errors** that may arise during the write operation.
Despite **exception handling** technically works, there is a better solution.

Context Managers

The better solution is using the `open()` function which is a **context manager**:

```
filename = 'somefile.txt'
with open(filename, 'w') as file:
    file.write('Something')

print(file.closed)
```

```
True
```

We use the `with` statement to create a context in which the file is opened. This ensures that the file is properly closed when the execution exits the `with` block, even if an exception is raised during the operation.

Generators and `yield` Statements

Consider a scenario where you are processing a large number of similar data sets. You need to process the data sets and perform certain operations on each of them. However, due to the large size, you can not load all the data sets in memory and pre-process them simultaneously.

Generators and `yield` Statements

Consider a scenario where you are processing a large number of similar data sets. You need to process the data sets and perform certain operations on each of them. However, due to the large size, you can not load all the data sets in memory and pre-process them simultaneously.

A possible solution is to only load a data set in memory when required and process only a single at a time. Even though we know what documents we will need, we do not load a resource until it is required. There is no need to retain the bulk of documents in memory when they are not in active use in our code. This is called **"lazy loading"**.

Generators and `yield` Statements

Consider a scenario where you are processing a large number of similar data sets. You need to process the data sets and perform certain operations on each of them. However, due to the large size, you can not load all the data sets in memory and pre-process them simultaneously.

A possible solution is to only load a data set in memory when required and process only a single at a time. Even though we know what documents we will need, we do not load a resource until it is required. There is no need to retain the bulk of documents in memory when they are not in active use in our code. This is called **”lazy loading”**.

Generators with the `yield` statement allow lazy-loading that improves the memory efficiency of Python code execution.

Generators and `yield` Statements

example:

```python
def load_documents(directory):
        for document_path in os.listdir(directory):
                with open(document_path) as _file:
                        yield _file

def preprocess_document(document):
        filtered_document = None
        # insert preprocessing code for the document
        # stored in filtered_document
        return filtered_document

directory = "docs/"
for doc in load_documents(directory):
        preprocess_document(doc)
```

In the above function, the `load_documents` function uses the `yield` keyword. The method returns an object of type `<class generator>`. When we iterate over this object, it continues execution from where the last `yield` statement is. Therefore, a single document is loaded and processed, improving Python code efficiency.

Generators and `yield` Statements

How Do Generators Work?

A generator function is defined like a regular function, but instead of using the `return` keyword, the `yield` keyword is used.

When calling a generator function, it returns a generator object. It can be iterated over using a loop or by calling `next()`.

When the `yield` statement is encountered, the function's state is saved, and the yielded value is returned to the caller. The function's execution pauses, but its local variables and state are retained.

When the generator's `next()` method is called again, execution resumes from where it was paused, and the function continues until the next `yield` statement.

When the function exits or raises a `StopIteration` exception, the generator is considered exhausted, and further calls to `next()` will raise `StopIteration`.

Caching

Caching is a technique used to improve application performance by temporarily storing results obtained by the program to reuse them if needed later.

To create a cache in Python, we can use the `@cache` **decorator** from the `functools` module.

example:

```python
# Using caching to speed up a function
import functools

@functools.cache
def square(n):
    print(f"Calculating square of {n}")
    return n * n

# Testing the cached function
print(square(4))  # Output: 16 (calculating square of 4)
print(square(4))  # Output: 16 (cached result, no recalculation)
```

Best Practices for Working With Large Datasets

Working with large datasets (as common in astronomy) can be a challenging task: it requires proper memory management to avoid running out of memory and to ensure the code runs efficiently. Here are some tips for working with large datasets and managing memory:

Use memory-efficient data structures: For example, instead of using Python's built-in list data structure, you can use NumPy arrays which are more memory efficient.

Use data sampling: It's often useful to work with a smaller subset of the data first while developing and testing the code. This can be done using techniques such as random sampling, which can help to reduce the amount of memory required to load and process the data.

Use lazy loading: Lazy loading is a technique for loading data into memory only when it is needed, rather than loading the entire dataset at once. This can help to reduce the amount of memory used by the program and make it more efficient.

Best Practices for Working With Large Datasets

Working with large datasets (as common in astronomy) can be a challenging task: it requires proper memory management to avoid running out of memory and to ensure the code runs efficiently. Here are some tips for working with large datasets and managing memory:

Use iterators and generators: Iterators and generators are a way to work with large datasets without loading the entire dataset into memory at once. They allow you to process the data one piece at a time, which can help to reduce the amount of memory used by the program.

Use disk-based storage: When working with large datasets that can't fit into memory, it's often useful to store the data on disk. Popular libraries such as HDF5 and Parquet allow you to store large datasets on disk and access it in a memory-efficient way.

Monitor memory usage: Regularly monitoring the memory usage of your program can help you identify and fix memory leaks, and optimize the memory usage of your program. Python provides libraries such as `memory_profiler` and `psutil` to monitor memory usage.

Summary: Optimizing Code Performance

Use built-in functions and libraries: Python has a lot of built-in functions and libraries that are highly optimized and can save you a lot of time and resources.

Use local global variables: Global variables can slow down your code, as they can be accessed from anywhere in the program.

Use list comprehensions instead of for loops: List comprehensions are faster than for loops because they are more concise and perform the same operations in fewer lines of code.

Avoid using recursion: Recursive functions can slow down your code because they take up a lot of memory. Instead, use iteration.

Use Cython to speed up critical parts of the code. It is a programming language that is a superset of Python but can be compiled into C, which makes it faster.

Use vectorized operations and broadcasting when performing calculations, it will make the code run faster.

Use multi-processing, multi-threading, or async IO to utilize multiple CPU cores and run multiple tasks simultaneously.

An Outlook: Parallelization

We have seen how to improve the performance of your code. We will look at some further techniques on how to do so during the next lectures.

In the next lecture we will see how to **use multi-processing and multi-threading** to parallelize code.