Programming for Engineers

Lecture 1 - basic data structures

Radoslav Škoviera

Table of contents

Basic data structures, intro to asymptotic complexity

Basic data structures (in Python)

Arrays (Python lists)

List is a build-in array-like data structure in Python. Unlike "traditional" arrays, Python lists are untyped. This makes their usage simpler but less efficient and sometimes "dangerous".

More efficient 'standard' array implementation can be found in the *NumPy* package, which we will discuss in a later lecture.

Creation

Creating empty lists can be done in two ways:

```
# Create an empty list
my_list = []
print(my_list)
my_list = list()
print(my_list)
```

```
[]
[]
```

Using the square brackets `[]` is the preferred way in Python. The function `list()` can also be used to to case any iterable to a list.

```
print(list("hello"))
```

```
['h', 'e', 'l', 'l', 'o']
```

```
# Create a list with some values
my_list = [4, 5, 6, 7, 8]
print(my_list)
```

```
[4, 5, 6, 7, 8]
```

```
# create a list of 10 copies of the same element
my_list = [7] * 10
print(my_list)
# useful for "initialization"
my_list = [0] * 10
print(my_list)
```

```
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Indexing & Assignment

Indexing is done using square brackets `[]`. The index is the position (zero-based) of the element in the list. Negative indices count from the end of the list.

```
my_list = [4, 5, 6, 7, 8]
print(my_list[0])  # first element
print(my_list[-1])  # last element
```

```
4
8
```

Slicing

Slicing allows to take a 'subset' of the list. The syntax is `my_list[start:end]` or `my_list[start:end:step]`. The result will include elements at indices starting from the `start` up to but not including the `end`. If the step is specified, this will be the increment (or decrement if negative) between indices (by default, the step is 1). If the slice should start from the beginning, the `start` value can be omitted. Likewise, if the slice should end at the end, the `end` value can be omitted.

```
my_list = [4, 5, 6, 7, 8]
print(my_list[0:2])  # first two elements
print(my_list[:2])  # first two elements, same as [0:2]
print(my_list[2:5])  # elements from index 2 to end (index 4)
print(my_list[2:])  # elements from index 2 to end (index 4), same as [2:5]
```

```
[4, 5]
[4, 5]
[6, 7, 8]
[6, 7, 8]
```

Negative

```
print(my_list[:-2])  # from beginning till the second to last index (excluding)
print(my_list[-2:])  # last two elements - from the second to last index till end
```

```
[4, 5, 6]
[7, 8]
```

Using the step

All of these are the same:

```python
print(my_list[0:5:2])  # every second element from the beginning to the end
print(my_list[0::2])  # every second element from the beginning to the end
print(my_list[::2])  # every second element from the beginning to the end
```

```
[4, 6, 8]
[4, 6, 8]
[4, 6, 8]
```

Negative step:

```python
print(my_list[::-1])  # reverse the list
```

```
[8, 7, 6, 5, 4]
```

…and any other combination you can imagine.

Deleting elements

There are several ways of how to remove elements from a list: splicing, 'popping' and deleting.

Splice (combine two sub-lists, excluding the removed element):

```python
my_list = [4, 5, 6, 7, 8]
my_list = my_list[:2] + my_list[3:]  # remove the element at index 2
print(my_list)
```

```
[4, 5, 7, 8]
```

Using the `pop` method:

```python
my_list = [4, 5, 6, 7, 8]
print(my_list.pop())  # remove the last element and return it
print(my_list)

print(my_list.pop(2))  # remove the element at index 2 and return it
print(my_list)
```

```
8
[4, 5, 6, 7]
6
[4, 5, 7]
```

Using the `del` keyword:

```python
my_list = [4, 5, 6, 7, 8]
del my_list[2]  # delete the element at index 2
print(my_list)
```

```
[4, 5, 7, 8]
```

There is also the `remove` method, which removes the first occurrence of the specified value.

```python
my_list = [4, 5, 6, 7, 8]
my_list.remove(5)
print(my_list)
my_list.remove(my_list[2])  # remove the element at index 2
```

```
[4, 6, 7, 8]
```

Deleting from the end of the array is typically faster than deleting from 'within' the array, which is still faster than deleting from the beginning of the array.

Inserting elements

Using the `append` method:

```python
my_list = [4, 5, 6, 7, 8]
my_list.append(9)
print(my_list)
```

```
[4, 5, 6, 7, 8, 9]
```

Using the `insert` method:

```
my_list = [4, 5, 7, 8]
my_list.insert(2, 6)  # insert 6 at index 2
print(my_list)
```

```
[4, 5, 6, 7, 8]
```

Inserting at the beginning of the array is typically faster than inserting in the middle or at the end of the array.

Extending an array with `extend`:

```
my_list = [4, 5, 7, 8]
my_list.extend([6, 9])
print(my_list)
```

```
[4, 5, 7, 8, 6, 9]
```

Or adding two lists with `+`:

```
my_list = [4, 5, 7, 8]
my_list = my_list + [6, 9]
print(my_list)
```

```
[4, 5, 7, 8, 6, 9]
```

Length, comparison and membership

Length of a list:

```
my_list = [4, 5, 7, 8]
print(len(my_list))
```

```
4
```

Remember the zero-based indexing:

```
n = len(my_list)
print(my_list[n - 1])
```

```
8
```

Comparison of lists (not very useful, usually you compare elements in a loop):

```
list_a = [1, 2, 3]
list_b = [4, 5, 6]

# comparison
print("\nComparison:")
print(list_a == list_b)  # equality
print(list_a != list_b)  # inequality

print("Equality is not strict but 'ordered'")
print("[1, 2, 3.0] == [1, 2, 3]: ", [1, 2, 3.0] == [1, 2, 3])  # True
print("[1, 2, 3] == [3, 2, 1]: ", [1, 2, 3] == [3, 2, 1])  # False

# Be (ALWAYS) aware of precision/numerical issues
print(
    "[1, 2, 3.000000000000000001] == [1, 2, 3]: ",
     [1, 2, 3.000000000000000001] == [1, 2, 3])  # True, depends on the precision
```

```
Comparison:
False
True
Equality is not strict but 'ordered'
[1, 2, 3.0] == [1, 2, 3]:  True
[1, 2, 3] == [3, 2, 1]:  False
[1, 2, 3.000000000000000001] == [1, 2, 3]:  True
```

Precision side-quest

```
print("3 == 3.00000001", 3 == 3.00000001)  # False
print("3 == 2.99999999", 3 == 2.99999999)  # False
print("1/3 == 0.33333333", 1/3 == 0.33333333)  # False
print("3.0000000000000001 == 3: ", 3.0000000000000001 == 3)  # True
print("2.9999999999999999 == 3: ", 2.9999999999999999 == 3)  # True
print("1/3 == 0.33333333333333333: ", 1/3 == 0.33333333333333333)  # True
```

```
3 == 3.00000001 False
```

```
3 == 2.99999999 False
1/3 == 0.33333333 False
3.000000000000000001 == 3:  True
2.99999999999999999 == 3:  True
1/3 == 0.33333333333333333333:  True
```

End of side-quest.

Inequality of lists:

```python
print("\nOrdering:")
print(list_a < list_b)  # Compares maximum value
print(list_a > list_b)
print(list_a <= list_b)
print(list_a >= list_b)
```

```
Ordering:
True
False
True
False
```

Membership checking:

```python
print("\nMembership:")
print(1 in list_a)  # membership
print(7 not in list_a)
```

```
Membership:
True
True
```

Other methods for lists

`index` - find the index of the first occurrence of the specified value

```python
my_list = [4, 5, 6, 7, 8]
print(f"Index of 5: {my_list.index(5)}")
```

```
Index of 5: 1
```

`count` - count the number of occurrences of the specified value

```python
my_list = [4, 5, 6, 5, 7, 7, 8, 7]
print(f"Count of 5: {my_list.count(5)}")
```

```
Count of 5: 2
```

`sort` - sort the list

```python
my_list = [6, 4, 7, 5, 3]
my_list.sort()
print(my_list)
my_list.sort(reverse=True)
print(my_list)
```

```
[3, 4, 5, 6, 7]
[7, 6, 5, 4, 3]
```

`reverse` - reverse the list

```python
my_list = [4, 5, 6, 7, 8]
my_list.reverse()
print(my_list)
```

```
[8, 7, 6, 5, 4]
```

`copy` - create a copy of the list

```python
my_list = [4, 5, 6, 7, 8]
not_a_copy = my_list  # not a copy, just a reference
not_a_copy.append(9)
print(my_list)  # 9 was added to the original list
my_list_copy = my_list.copy()  # make a copy
my_list_copy.append(10)
print(my_list_copy)  # 10 was added to the copy
print(my_list)  # the original list is not affected
```

```
[4, 5, 6, 7, 8, 9]
[4, 5, 6, 7, 8, 9, 10]
[4, 5, 6, 7, 8, 9]
```

Buuuut, `copy` is not a *deep* copy:

```
my_dict = {"a": 1, "b": 2}
my_list = [[4, 5, 6], my_dict]
my_list_copy = my_list.copy()
my_list_copy[1]["a"] = 3
print(my_list)
```

```
[[4, 5, 6], {'a': 3, 'b': 2}]
```

Although, you shouldn't be using multi-typed lists anyway.

Heterogeneous lists ("jagged" and multi-typed)

It is best to avoid using heterogeneous lists (though sometimes, they are handy).

```
# Jagged nested list
my_list = [[1, 2, 3], [4, 5], [6]]
print(my_list)
```

```
[[1, 2, 3], [4, 5], [6]]
```

```
# Multi-typed
my_list = [1, 2, 3, "hello", [4, 5, 6]]
print(my_list)
```

```
[1, 2, 3, 'hello', [4, 5, 6]]
```

Advanced creation

The `range` function creates a "range" object containing a sequence of integers from 0 to some number, excluding the last number. This can be converted to a list using the `list()` function.

```python
# Create a list of the first 5 integers
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

The `range` function allows you to specify the starting number and the step size. The general syntax is `range(start, stop, step)`.

```python
# Numbers from 5 to 10
print(list(range(5, 10)))
```

```
[5, 6, 7, 8, 9]
```

```python
# Create a list of even numbers from 0 to 10
print(list(range(0, 10, 2)))
```

```
[0, 2, 4, 6, 8]
```

List comprehensions are a concise way to create lists. However, they are technically equivalent to a for loop.

```python
my_list = [x * 2 for x in range(5)]
print(my_list)
```

```
[0, 2, 4, 6, 8]
```

Which is equivalent to:

```python
my_list = []
for x in range(5):
    my_list.append(x * 2)
print(my_list)
```

```
[0, 2, 4, 6, 8]
```

The syntax for list comprehensions is:

```
[<expression> for <item> in <iterable>]
```

Comprehension, in general, are 'one-liner' creators of collections of objects. They can in some cases make the code more readable and in some cases much less readable. Basic Python datatypes support comprehensions.

Comprehension with conditions:

```
[<expression> for <item> in <iterable> if <condition>]
```

`<expression>` will be added to the list only if `<condition>` is `True`.

```
odd_numbers = [x for x in range(10) if x % 2 == 1]
print(odd_numbers)
```

```
[1, 3, 5, 7, 9]
```

Comprehension with conditions and if-else:

```
[<expression_one> if <condition> else <expression_two> for <item> in <iterable>]
```

`<expression_one>` will be added to the list if `<condition>` is `True`. Otherwise, `<expression_two>` will be added to the list (`<condition>` is `False`).

```
even_or_negative_numbers = [x if x % 2 == 1 else -x for x in range(10)]
print(even_or_negative_numbers)
```

```
[0, 1, -2, 3, -4, 5, -6, 7, -8, 9]
```

Dictionaries

Dictionaries are like lookup tables. They are implemented using a hash table, which means accessing an element is very fast.

Creation

Empty dictionarys can be created using the `dict()` function or with `{}` (preferred way).

```
empty_dict = dict()
also_empty_dict = {}
```

Initialization:

```
my_dict = {
    "a": 1,
    "b": 2,
}
print(my_dict)
```

```
{'a': 1, 'b': 2}
```

Deleting and inserting

Insert

```
my_dict = {}
my_dict["a"] = 1
my_dict["b"] = 2
print(my_dict)
```

```
{'a': 1, 'b': 2}
```

Delete

```
my_dict = {"a": 1, "b": 2}
del my_dict["a"]
print(my_dict)
```

```
{'b': 2}
```

Retrieving value

```
my_dict = {"a": 1, "b": 2}
print(my_dict["a"])
print(my_dict.get("a"))  # safe retrieval, return None if "a" not in dictionary
print(my_dict.get("c"))  # safe retrieval, return None if "a" not in dictionary
print(my_dict.get("a", -1))  # default value
print(my_dict.get("c", -1))  # default value
```

```
1
1
None
1
-1
```

Membership

```
my_dict = {"a": 1, "b": 2}
print("a" in my_dict)
print("c" in my_dict)
```

```
True
False
```

Advanced creation

Dictionary comprehension:

```
{x: x**2 for x in range(5)}
```

Hashmaps

The `hash()` function returns the hash value of an object - seemingly arbitrary but for the same input consistent value (headache warning: the consistency holds only for the current "session"!).

```
print(hash("hello"))
print(hash("hello") == hash("hello"))
```

```
8721379684979654669
True
```

Array access, given an index is *fast* (constant time), however, finding a specific value is slow. Hashmaps solve this by encoding the value into an index.

```
def hash_into_index(value, total_items):
    return hash(value) % total_items

print(hash_into_index("hello", 10))
```

9

```python
class MyHashmap:
    def __init__(self, total_items):
        self.total_items = total_items
        self.keys = [None] * total_items
        self.values = [None] * total_items

    def __setitem__(self, key, value):  # magic to allow indexed assignment
        index = hash_into_index(key, self.total_items)
        self.keys[index] = key
        self.values[index] = value

    def __getitem__(self, key):  # magic to allow indexing
        index = hash_into_index(key, self.total_items)
        return self.values[index]

    def __contains__(self, key):  # magic to allow the "in" keyword
        index = hash_into_index(key, self.total_items)
        return self.keys[index] == key

    def __iter__(self):  # magic to allow the for-each loop
        for key, value in zip(self.keys, self.values):
            if key is not None:
                yield key, value

    def print(self):
        for key, value in self:
            print(f"{key}: {value}")

hashmap = MyHashmap(10)
hashmap["hello"] = "world"
print(hashmap["hello"])
print("hello" in hashmap)

print("Contents of our hashmap:")
hashmap["water"] = "world"
hashmap["hellno"] = "word"
hashmap["pi"] = 3.14
hashmap["e"] = 2.718
hashmap[9] = "number hashes are the same as the number itself"
hashmap.print()
```

```
world
True
Contents of our hashmap:
pi: 3.14
e: 2.718
9: number hashes are the same as the number itself
```

Obviously, this is not a very good implementation of a hashmap: 1) Collisions (multiple keys hash to the same index) 2) Memory usage (None values for empty slots)

Tuples & Sets

Besides lists, Python also has tuples and sets. These can also store multiple values but offer different functionalities.

Creation

Tuples

A tuple is an immutable list. Tuples are created using the parentheses `()`. Similarly, the `tuple()` function can be used to convert an iterable to a tuple.

Empty tuple:

```python
empty_tuple = tuple()  # kinda pointless - tuples cannot be changed
empty_tuple = ()  # kinda pointless - tuples cannot be changed
```

Single element tuple:

```python
single_element_tuple = (1,)  # mind the comma! (1) is just 1 in brackets
print((2) * 10)  # just `2 * 10` => 20
print((2,) * 10)  # tuple of 10 2s
```

```
20
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

Creating tuples:

```
my_tuple = (1, 2, 3)
print(my_tuple)

# Create a tuple from a list
my_list = [4, 5, 6]
my_tuple = tuple(my_list)
print(my_tuple)
```

```
(1, 2, 3)
(4, 5, 6)
```

Tuple comprehension:

```
my_tuple = (x * 2 for x in range(5))
print(my_tuple)
```

```
<generator object <genexpr> at 0x7f31d00e5a40>
```

Oops, that's not what we wanted! Actual tuple comprehension (we might talk about generators later):

```
my_tuple = tuple(x * 2 for x in range(5))
print(my_tuple)
```

```
(0, 2, 4, 6, 8)
```

Sets

A set is an unordered collection of unique elements. Sets are created using the `{<iterable>}`. Similarly, the `set()` function can be used to convert an iterable to a set.

Empty set:

```
empty_set = set()
```

Careful, `{}` will not create an empty set but an empty dictionary! (see below)

Creating sets:

```python
# Create a set
my_set = {1, 2, 3}
print(my_set)

# Create a set from a list
my_list = [4, 5, 6]
my_set = set(my_list)
print(my_set)
my_set.add(7)  # Add an element to the set
print(my_set)
```

```
{1, 2, 3}
{4, 5, 6}
{4, 5, 6, 7}
```

Adding elements to a set is an idempotent operation - adding the same element twice does not change the set.

```python
my_set = {1, 2, 3}
print("my_set: ", my_set)
my_set.add(4)
print("my_set after adding 4: ", my_set)
for i in range(100):
    my_set.add(4)
print("my_set after adding 4 one hundred times: ", my_set)
```

```
my_set:  {1, 2, 3}
my_set after adding 4:  {1, 2, 3, 4}
my_set after adding 4 one hundred times:  {1, 2, 3, 4}
```

Set comprehension:

```python
my_set = {int(x / 2) for x in range(15)}
print(my_set)
```

```
{0, 1, 2, 3, 4, 5, 6, 7}
```

Sets are not immutable but they do not support assignment. Rather, you can add or remove elements.

```
my_set.add(10)  # The set can be changed.
my_set[0] = 10  # This will raise an error!!!
```

Tuple differences with Lists

Main difference is that tuples are immutable. They cannot be changed once created. Some operations are faster than on lists.

```
my_tuple = (1, 2, 3)
my_tuple[1] = 4  # This will raise an error!!!
```

Tuples are used as the return type in case of functions returning multiple values.

```
def my_function():
    n = 5
    l = [i for i in range(n)]
    return l, n

print(my_function())
```

```
([0, 1, 2, 3, 4], 5)
```

Tuples are "safe" to pass around because of their immutability.

```
def my_function(iterable_input):
    result = []
    for i in iterable_input:
        result.append(i * 2)
    # sneakily change the input:
    iterable_input[0] = 10
    return result

my_list = [1, 2, 3]  # precious data we don't want to change
print(f"My list before running my_function: {my_list}")
print(my_function(my_list))
print(f"My list after running my_function: {my_list}")

my_tuple = (1, 2, 3)  # precious data we don't want to change
try:
    print(my_function(my_tuple))
```

```python
except TypeError:  # this will catch the error raised
    # when the function tries to change the tuple
    print("Aha! The function tried to change my tuple!")
```

```
My list before running my_function: [1, 2, 3]
[2, 4, 6]
My list after running my_function: [10, 2, 3]
Aha! The function tried to change my tuple!
```

Set differences with Lists

Sets are unordered and do not allow duplicates. They are implemented as 'hash set' and thus certain operations are very efficient (e.g. membership checking, union, intersection, etc.).

```python
my_list = [1, 2, 2, 3, 3, 4, 5, 5]
my_set = set(my_list)
print(my_set)
print(3 in my_set)  # True
print(6 in my_set)  # False
my_set.add(5)  # Added element already in list, nothing happens
print(my_set)
```

```
{1, 2, 3, 4, 5}
True
False
{1, 2, 3, 4, 5}
```

Set specific operations:

```python
print({1, 2, 3} | {4, 5, 6})  # Union
print({1, 2, 3} & {3, 4, 5})  # Intersection
print({1, 2, 3, 4} - {1, 4, 5})  # Difference
print({1, 4, 5} - {1, 2, 3, 4})  # Difference (reversed set order)
print({1, 4, 5} ^ {1, 2, 3, 4})  # Symmetric Difference (XOR)
```

```
{1, 2, 3, 4, 5, 6}
{3}
{2, 3}
{5}
{2, 3, 5}
```

Since tuple are immutable, unlike lists or sets, they can be used as keys in a dictionary.

```
my_tuple_dict = {
    (1, 2): 3,
    (4, 5): 6
}

print(my_tuple_dict[(1, 2)])
```

```
3
```

This would throw an error:

```
my_list_dict = {
    [1, 2]: 3,
    [4, 5]: 6
}
```

Basic operations with data structures

Looping through iterables

Indexed loop

```
my_list = list('alphabet')
n = len(my_list)
for i in range(n):
    value = my_list[i]
    print(f"Value at index {i}: {value}")
```

```
Value at index 0: a
Value at index 1: l
Value at index 2: p
Value at index 3: h
Value at index 4: a
Value at index 5: b
Value at index 6: e
Value at index 7: t
```

‘For each’ loop

```python
for value in my_list:
    print(f"Value: {value}")
```

```
Value: a
Value: l
Value: p
Value: h
Value: a
Value: b
Value: e
Value: t
```

Enumerate

The `enumerate()` function returns a tuple of the index and the value at that index. This is equivalent to the indexed loop but ‘nicer’ (no need to explicitly extract the value and index).

```python
for index, value in enumerate(my_list):
    print(f"Value at index {index}: {value}")
```

```
Value at index 0: a
Value at index 1: l
Value at index 2: p
Value at index 3: h
Value at index 4: a
Value at index 5: b
Value at index 6: e
Value at index 7: t
```

While loop

The `while` loop is useful in some special cases (e.g., growing lists - although, this can be dangerous).

```python
from random import random

my_list = list(range(10))
n = len(my_list)
i = 0
```

```
while i < len(my_list):
    value = my_list[i]
    if random() < 1 / (i - value + 1.5):
        my_list.append(value)
    i += 1

print(my_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 2, 3, 4, 5, 6, 7, 8, 9]
```

More typical is to use infinite loops with `while True` and `break` statements.

Looping through sets and tuples

The same looping "techniques" work for sets and tuples.

```
my_set = {1, 2, 3}
for i, value in enumerate(my_set):
    print(f"Value: {value}")
```

```
Value: 1
Value: 2
Value: 3
```

```
my_tuple = (1, 2, 3)
for value in my_tuple:
    print(f"Value: {value}")
```

```
Value: 1
Value: 2
Value: 3
```

Looping through dictionaries

Most common way to iterate over a dictionary is to use the `items()` method, which will return (key, value) pairs.

```python
import string

alphabet_dict = {
    k: v for k, v in zip(string.ascii_letters, range(26))
     if v < 12 and v % 2 == 0}
for key, value in alphabet_dict.items():
    print(f"The letter {key} is at position {value + 1} in the alphabet.")
```

```
The letter a is at position 1 in the alphabet.
The letter c is at position 3 in the alphabet.
The letter e is at position 5 in the alphabet.
The letter g is at position 7 in the alphabet.
The letter i is at position 9 in the alphabet.
The letter k is at position 11 in the alphabet.
```

There are also `keys()` and `values()` methods that return the keys and values respectively.

```python
for key in alphabet_dict.keys():
    print(f"The letter {key} is at position "
    f"{alphabet_dict[key] + 1} in the alphabet.")
    # Yes, you can break strings in Python like that...
```

```
The letter a is at position 1 in the alphabet.
The letter c is at position 3 in the alphabet.
The letter e is at position 5 in the alphabet.
The letter g is at position 7 in the alphabet.
The letter i is at position 9 in the alphabet.
The letter k is at position 11 in the alphabet.
```

```python
positions = []
for value in alphabet_dict.values():
    positions.append(value + 1)
print(f"The dictionary contains letters at these positions: {positions}")
```

```
The dictionary contains letters at these positions: [1, 3, 5, 7, 9, 11]
```

2D Arrays (Matrices)

Simply "stack" lists inside of a list (nested list).

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix)
also_matrix = [[j + i * 3 for j in range(3)] for i in range(3)]
print(also_matrix)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

Accessing elements

`matrix[row][column]`:

```
print(matrix[0][1])  # 0th row, 1st column
print(matrix[1][0])  # 1st row, 0th column
```

```
2
4
```

Looping through matrices

For each row in the matrix, loop through each element in the row.

```
for row in matrix:
    for value in row:
        print(value)
```

```
1
2
3
4
5
6
```

```
7
8
9
```

Enumerate the rows and columns.

```
for row_index, row in enumerate(matrix):
    for column_index, value in enumerate(row):
        print(f"Value at row {row_index} and column {column_index}: {value}")
```

```
Value at row 0 and column 0: 1
Value at row 0 and column 1: 2
Value at row 0 and column 2: 3
Value at row 1 and column 0: 4
Value at row 1 and column 1: 5
Value at row 1 and column 2: 6
Value at row 2 and column 0: 7
Value at row 2 and column 1: 8
Value at row 2 and column 2: 9
```

Assignment to a matrix:

```
for row_index, row in enumerate(matrix):
    for column_index, value in enumerate(row):
        matrix[row_index][column_index] = value**2
print(matrix)
```

```
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

Types and comparisons

Checking types of variables is very useful, especially in Python that allows dynamic typing. Many operations are defined for different data types. For example, `1 * 2` is fine, and so is `[1] * 2` and `"1" * 2` but they result in different outcomes. It is often important to assert that the variables are of the expected type (use `if` or `assert`).

It is or it is not

The `is` operator can be used to check if two objects are the same object.

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
d = a[:]
e = a.copy()
# the elements have the same value but `a` is not the same object as `b`
print(a is b)

# `a` and `c` reference the same object
print(a is c)

# `d` got the values from `a` but it is still a different object
# (different memory location)
print(a is d)

# copy makes a new object with the same values
# (essentially, similar process to defining `d`)
print(a is e)
```

```
False
True
False
False
```

It is not?

```
print(a is not b)  # True
print(a is not c)  # False
print(a is not d)  # True
print(a is not e)  # True
```

```
True
False
True
True
```

Some ‘objects’ are the same but it makes no sense to compare them using `is`. Although, `is` will get you a strict equality.

```
# this will get you a syntax warning
print(1 is 1.0)  # False, because float != int
print(1.0 is 1.0)  # True
print((1, 2, 3) is (1, 2, 3))  # Unexpectedly, True but also not advised;
#just use equality `==`
```

```
False
True
True
```

```
<>:2: SyntaxWarning:

"is" with 'int' literal. Did you mean "=="?

<>:3: SyntaxWarning:

"is" with 'float' literal. Did you mean "=="?

<>:4: SyntaxWarning:

"is" with 'tuple' literal. Did you mean "=="?

<>:2: SyntaxWarning:

"is" with 'int' literal. Did you mean "=="?

<>:3: SyntaxWarning:

"is" with 'float' literal. Did you mean "=="?

<>:4: SyntaxWarning:

"is" with 'tuple' literal. Did you mean "=="?

/tmp/ipykernel_61522/4199038124.py:2: SyntaxWarning:

"is" with 'int' literal. Did you mean "=="?

/tmp/ipykernel_61522/4199038124.py:3: SyntaxWarning:

"is" with 'float' literal. Did you mean "=="?
```

```
/tmp/ipykernel_61522/4199038124.py:4: SyntaxWarning:

"is" with 'tuple' literal. Did you mean "=="?
```

For number type check, see below.

type

Use the `type(<object>)` build-in function to get the type of an object.

```python
print(type(1))  # <class 'int'>
print(type(1.0))  # <class 'float'>

my_list = [1, 2, 3]
print(type(my_list))  # <class 'list'>
print(type(tuple(my_list)))  # <class 'list'>
```

```
<class 'int'>
<class 'float'>
<class 'list'>
<class 'tuple'>
```

isinstance

The `isinstance(<object>, <type>)` build-in function checks whether the object is an instance of the type.

```python
print(isinstance([1, 2, 3], list))  # True
print(isinstance([1, 2, 3], tuple))  # False
print(isinstance(1, int))  # True
print(isinstance(1, float))  # False
```

```
True
False
True
False
```

issubclass

This will become handy much later, when you will be working with classes. However, there is some use with basic datatypes.

```
my_integer = 3
print(issubclass(type(my_integer), int))  # True
print(issubclass(type(my_integer), float))  # False
print(issubclass(type(my_integer), (int, float)))  # True
```

```
True
False
True
```

Being assertive in your statements (programmatically speaking)

Potentially unsafe code:

```
def multiply_two_numbers_unchecked(a, b):
    return a * b

print(multiply_two_numbers_unchecked(3, 2))  # 6 is fine
print(multiply_two_numbers_unchecked(3.0, 2))  # 6.0 is fine
print(multiply_two_numbers_unchecked("3", 2))  # 33 !?
```

```
6
6.0
33
```

Code with run-time type checking and assertions:

```
def multiply_two_numbers(a, b):
    assert isinstance(a, (int, float)), "'a' must be a number! "
    f"But it was: {type(a)}"
    assert isinstance(b, (int, float)), "'b' must be a number! " f"But it was: {type(b)}"
    # Alternatively, this will also work:
    assert issubclass(type(my_integer), (int, float)), "'a' must be a number! "
    f"But it was: {type(a)}"
    return a * b
```

```
print(multiply_two_numbers(3, 2))
print(multiply_two_numbers(3.0, 2))
try:
    print(multiply_two_numbers("3", 2))  # this will cause an error
except AssertionError as e:
    print(e)
```

```
6
6.0
'a' must be a number!
```

More info about basic types and comparison: https://docs.python.org/3/library/stdtypes.html

But, maybe, don't be too assertive?

`assert` is fine for testing and debugging, but not for production code! There are better, safer ways of handling incorrect inputs (`try...except` + `warn`/`log.error` and handle the erroneous state "gracefully"). Essentially, production code should never halt!!!

```
from typing import Union, Optional

def nice_multiply_two_numbers(
    a: Union[int, float],
    b: Union[int, float],
    raise_error: bool = True) -> Optional[Union[int, float]]:
    if not isinstance(a, (int, float)):
        if raise_error:
            raise TypeError(f"'a' must be a number! But it was: {type(a)}")
        else:
            return None  # does not have to be explicit
    if not isinstance(b, (int, float)):
        if raise_error:
            raise TypeError(f"'b' must be a number! But it was: {type(b)}")
        else:
            return None  # does not have to be explicit
    return a * b

print("Using type error & try...except:")
try:
    print(nice_multiply_two_numbers(3, 2))
    print(nice_multiply_two_numbers(3.0, 2))
    print(nice_multiply_two_numbers("3", 2))  # this will cause an error
except TypeError as e:
    print(f"One of the inputs had an incorrect type. See the error:\n{e}")

print("\nUsing None return type:")
result = nice_multiply_two_numbers("3", 2, raise_error=False)
if result is None:
    print("One of the inputs had an incorrect type.")
else:
    print(result)
```

```
Using type error & try...except:
6
6.0
One of the inputs had an incorrect type. See the error:
'a' must be a number! But it was: <class 'str'>

Using None return type:
One of the inputs had an incorrect type.
```

Algorithmic complexity

Efficiency of algorithms matter.

Basic concepts

Runtime of an algorithm typically depends on the size of the input. For example, time to loop once through a list will linearly increase with the size of the list. Thus, if accessing an element of a list takes 1ns, looping through a list of N elements will take N * 1ns. However, on a different computer, the element access time might be 10ns. Still, the loop will take N * 10ns. Therefore, we simply say the runtime 'scales' with N.

Searching for a specific item in a list requires looping through the list. However, the item might be at the beginning of the list, thus the search time will be 1 (x access time). If the idem is at the end of the list, the search time will be N. On average, we can expect the search time to be N/2. We are, however, typically interested in the worst-case scenario, and for simplicity, ignore any constants. Therefore, we simply say that the lookup time 'scales' with N, just as with the 'full' loop. (There is typically some overhead and randomness, so we are not interested in precise numbers.)

Asymptotic complexity is a way to describe the runtime of an algorithm as a function of the input size. We can use this to compare the performance of different algorithms - the efficiency of their implementation. For example, we can compare different sorting algorithms.

How to "profile" in Python

We can use the `timeit` module to profile the runtime of an algorithm. Usage:

```python
import timeit
def my_function():
    s = 0
    for i in range(10000):
        s += i
    return s
iterations = 10
timer = timeit.Timer(stmt=my_function)
avg_time = timer.timeit(number=iterations) / iterations
print(f"Average run time: {avg_time}")
```

```
Average run time: 0.00046374450002986125
```

In Jupyter notebooks or IPython, we can use the `%timeit` magic command instead:

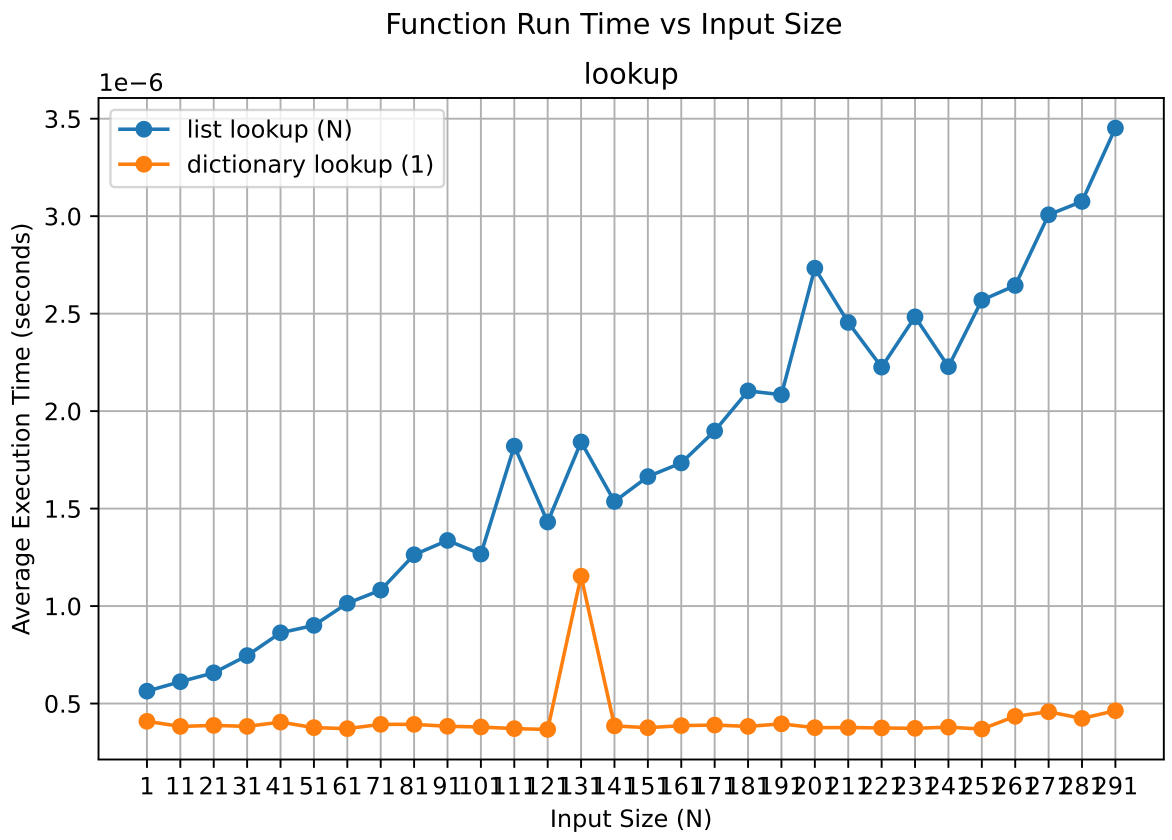
```
%timeit my_function()
```

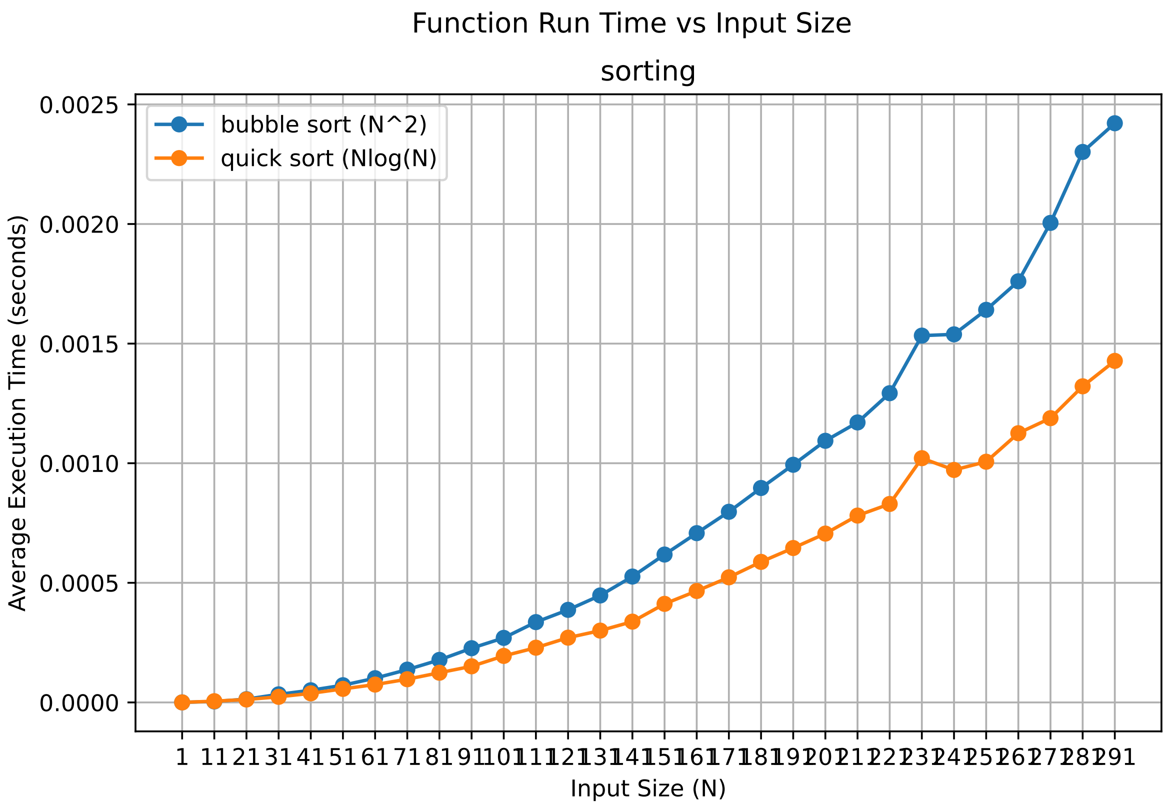
```
320  s ± 7.87  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

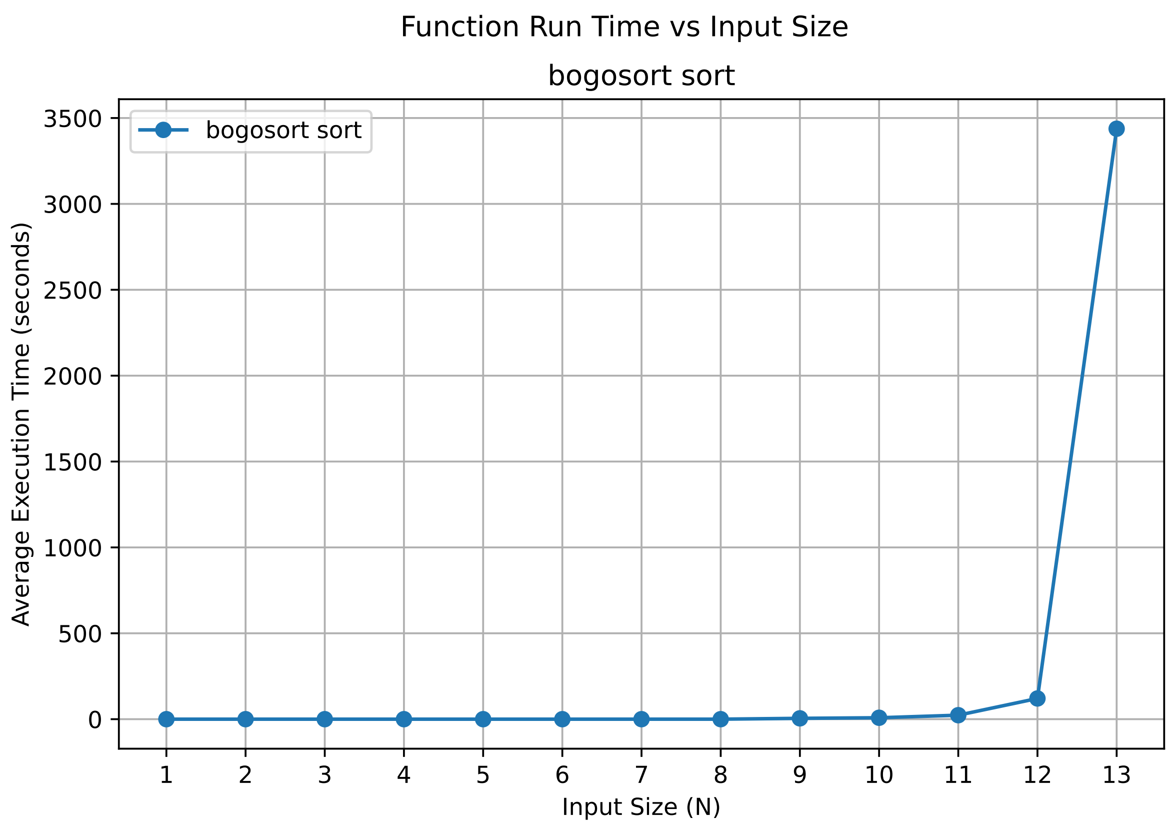
This will run the function and time it.

There are also better, more advanced tools for profiling (e.g. `cProfile`) but we will manage with timeit for now.

Run time examples




Function Run Time vs Input Size
sorting
bubble sort (N^2)
quick sort (Nlog(N)
Average Execution Time (seconds)
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
1
11
21
31
41
51
61
71
81
91
101
111
121
131
141
151
161
171
181
191
201
211
221
231
241
251
261
271
281
291
Input Size (N)



Pre-allocation

Pre-allocation of arrays is (slightly) faster than iterative appending. Although, in Python, both are relatively slow. Depending on the task, list comprehension may be more efficient. In general, if the appending overhead is insignificant, it will not have significant impact on runtime whether pre-allocation is used. However, with large loops it might cause memory issues and pre-allocation will be important with more efficient array implementations (e.g., NumPy arrays).

```python
import numpy as np

def my_simple_function(x):
    return x

def my_complex_function(x):
    return np.sqrt(np.log(x + 1)**min(1e2,  np.exp(np.log(x + 1e-10))**0.33))

def list_append(n, func):
    a = []
    for i in range(n):
        a.append(func(i))
    return a

def list_preallocate(n, func):
    a = [0] * n
    for i in range(n):
        a[i] = func(i)
    return a

def list_comprehension(n, func):
    return [func(i) for i in range(n)]

N = 100
print("Evaluating my_simple_function")
%timeit list_append(N, my_simple_function)
%timeit list_preallocate(N, my_simple_function)
%timeit list_comprehension(N, my_simple_function)

print("Evaluating my_complex_function")
%timeit list_append(N, my_complex_function)
%timeit list_preallocate(N, my_complex_function)
%timeit list_comprehension(N, my_complex_function)
```

```
Evaluating my_simple_function
3.89  s ± 88.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
3.34  s ± 55.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
3.61  s ± 128 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Evaluating my_complex_function
322  s ± 23.8  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
299  s ± 14.4  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
307  s ± 17.4  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Using the right data structures

Python lists are fairly universal. However, they come with some overhead and thus are not always the best structure. In some cases, it is even worth to cast lists to sets or dictionaries.

```
N = 100
# Generate list of randomly shuffled squares of numbers
```

```python
l = (np.random.permutation(N)**2).tolist()

# Create a dictionary from the list
d = {x: x**2 for x in l}

# Create a set from the list
s = set(l)

# We want to find the index of a square of a number in the list
number_of_interest = int(N / 2)**2
print("Lookup time for list")
%timeit l.index(number_of_interest)
print("Lookup time for dictionary")
%timeit d.get(number_of_interest)
print("Lookup time for set")
%timeit s.intersection({number_of_interest})
print("---")

# Now we want to simply see if the number is in the list
print("Membership check time for list")
%timeit number_of_interest in l
print("Membership check time for dictionary")
%timeit number_of_interest in d
print("Membership check time for set")
%timeit number_of_interest in s

print("---")
print("'Fair' lookup time for dictionary from list")
%timeit {x: x**2 for x in l}.get(number_of_interest)
print(f"'Fair' membership check time for dictionary from list")
%timeit number_of_interest in {x: x**2 for x in l}
```

```
Lookup time for list
681 ns ± 21.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Lookup time for dictionary
39.8 ns ± 0.88 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
Lookup time for set
138 ns ± 1.87 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
---
Membership check time for list
472 ns ± 16.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Membership check time for dictionary
```

```
34 ns ± 0.758 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
Membership check time for set
29.1 ns ± 1.31 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
---
'Fair' lookup time for dictionary from list
6.63  s ± 102 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
'Fair' membership check time for dictionary from list
6.7  s ± 68.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```