bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

COMPUTER SCIENCE

Data Structures II

Objectives

- Revise the structures we've seen so far
- Discuss certain data structures in depth
- Distinguish between data structures in C and Python
- Discover List comprehension

What have we seen so far?

- `List`
- `Tuple`
- `NamedTuple`
- `Dictionary`
- `Set`
- `Queue and LiFO`

Can you name what is common and what is unique to each?

How are they created and when to use them?

Is there more available in the Python standard library?

Arrays?

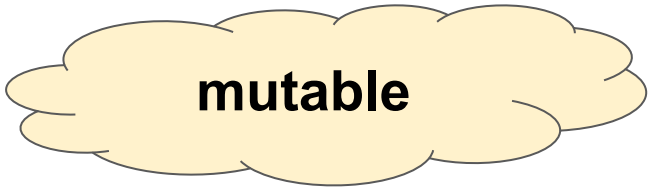
Access to indexed element is constant O(1) for each element.

- You would have encountered arrays in C
  - Don’t we have arrays in Python?
  - Fundamental data structure that we see in most programming language
- Arrays are considered contiguous data structures
  - They store information in adjoining blocks of memory
  - Efficiently located, available via index
  - Fixed size

  - Parking lot analogy, for specific vehicles: -> type

Python includes several “array-like” structures in its standard library. Many other libraries offer extensions to the built-ins.

“Arrays” In Python: `list`



- List are implemented as dynamic arrays
  - Elements can be added and removed and the list will automatically adjust memory allocation
- Can hold arbitrary elements
  - C arrays are typed!
  - Not in Python, here you can put everything in that you like, which means that elements are not as “tightly packed”, i.e. needs more space!

Simple `list` Exercises

[2]

```
my_array = ["one", "two", "three"]
print(my_array[0])

# nice print
print(my_array)

# Lists are mutable:
my_array[1] = "hello"
print(my_array)

del my_array[1]
print(my_array)

# Lists can hold arbitrary data types:
my_array.append(41)
print(my_array)
```

```
/Users/bianca.schoenphelan/D
['one', 'two', 'three']
['one', 'hello', 'three']
['one', 'three']
['one', 'three', 41]
```

List comprehensions

- Comprehensions are constructs that allow sequences to be built from other sequences
- Python 2.0 introduced list comprehensions
- Python 3.0 introduced dictionary and set comprehensions

Different ways of creating lists in Python

1. Using a for loop

[3]

```
squares = []
for i in range(10):
    squares.append(i * i)

print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. Using map()
3. Comprehension: elegant re-write of the loop

List comprehension

Defines the list and its contents at the same time.

[3]

```
squares = [i * i for i in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
new_list = [expression for member in iterable]
```

List comprehension

[3]

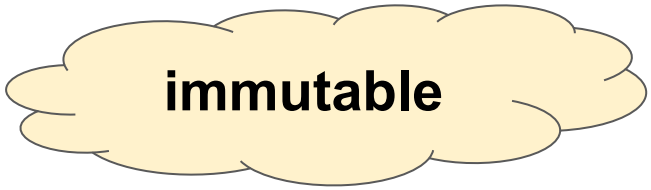
```
new_list = [expression for member in iterable]
```

Expression: any valid expression that returns a value, such as the member itself, a function call

Member: Object or value in the list or iterable

Iterable: List, set, sequence, generator or other object that can return its values one at a time

“Arrays” in Python: `tuple`



- Elements cannot be added or removed dynamically
- Can hold arbitrary data types
  - Powerful
  - But also not as memory efficient as typed arrays
- Item assignment not allowed
- Item deletion not allowed
- You can add elements, but this results in a copy of the tuple and not the original tuple

Simple **tuple** Exercises

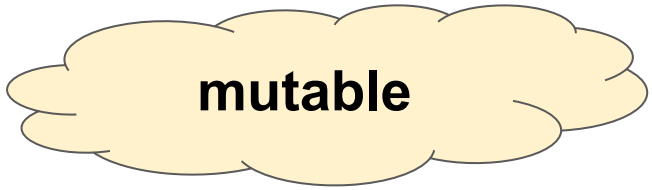
[2]

```
my_array = ("one", "two", "three")
print(my_array[0])

# easy printing
print(my_array)

# Tuples are immutable:
my_array[1] = "hello"  #causes an TypeError message
del my_array[1] #causes an TypeError message

# Tuples can hold arbitrary data types:
# (Adding elements creates a copy of the tuple)
print(id(my_array)) #original: 140527352248512
my_array = my_array + (41,)
print(my_array) # prints ('one', 'two', 'three', 41)
print(id(my_array)) #copy of same name: 14052736325918
```

```
one
('one', 'two', 'three')
140527352248512
('one', 'two', 'three', 41)
140527363259184
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/tutorial.py", line 33, in <module>
    my_array[1] = "hello"  #causes an TypeError message
TypeError: 'tuple' object does not support item assignment
```

Arrays in Python: `array.array`



- Based on c-style arrays
- Needs to be imported `import array`
- Behave similarly to lists because they are mutable, but they are typed
- Elements are tightly packed, according to their type
  - Memory saving container
- Code looks very similar to list, so exchanging one for the other is intuitive

Python `array.array` Exercise

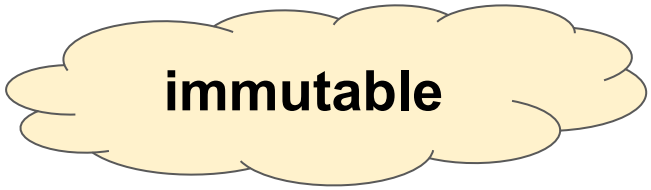
The data type

[2]

```
import array
my_array = array.array("f", (1.0, 1.5, 2.0, 2.5))
print(my_array[1])

# nice print
print(my_array)

# Arrays are mutable:
my_array[1] = 41.0
print(my_array)

del my_array[1]
print(my_array)

my_array.append(42.0)
print(my_array)


# Arrays are "typed":
my_array[1] = "hello" #causes an error message
```

```
/Users/bianca.schoenphelan/Documents/OOP_Class/Code/venv/bin/python /Users/bianca.schoenphelan/
1.5
array('f', [1.0, 1.5, 2.0, 2.5])
array('f', [1.0, 41.0, 2.0, 2.5])
array('f', [1.0, 2.0, 2.5])
array('f', [1.0, 2.0, 2.5, 42.0])
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/tutorial.py", line 64, in <module>
    my_array[1] = "hello" #causes an error message
TypeError: must be real number, not str

Process finished with exit code 1
```

“Arrays” in Python: `str`



- Stores textual data as immutable sequence of unicode characters
  - An immutable array of characters
- Recursive data structure:
  - Each element within is of type `str` of length=1
- Tightly packed
- Modification requires creating a copy
- Closest equivalent to a mutable string is storing individual characters in a list

`str` Exercises

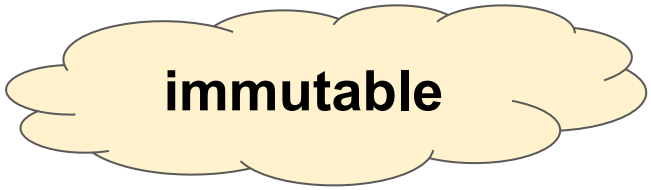
```
my_array = "abcd"
print(my_array[1])

# nice print
print(my_array)

# Strings are immutable:
my_array[1] = "e" #causes an error

del my_array[1] #causes an error

# Strings can be unpacked into a list to
# get a mutable representation:
my_array_lst = list("abcd")
print(my_array_lst)

from_lst_to_str = "".join(list("abcd"))

# Strings are recursive data structures:
print(type(from_lst_to_str))

print(type(from_lst_to_str[0]))
```

```
b
abcd
['a', 'b', 'c', 'd']
<class 'str'>
<class 'str'>
```

```
eback (most recent call last):
le "/Users/bianca.schoenphelan/Documents/OOP_Class/Code
 my_array[1] = "e" #causes an error
Error: 'str' object does not support item assignment
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/
    del my_array[1] #causes an error
TypeError: 'str' object doesn't support item deletion
```

“Arrays” in Python: `bytes`



- Immutable sequence of single bytes or integers between 0 and 255 (incl. both)
- Space efficient
- Very similar to `str` just store a different type
- But there is a dedicated mutable byte array data type called `bytearray`
  - Copying back from bytearray to bytes takes O(n) time

`bytes` Exercises

[2]

```
my_array = bytes((0, 1, 2, 3))
print(my_array[1])

# Bytes literals have their own syntax:
print(my_array)

my_array = b"\x00\x01\x02\x03"

# Only valid `bytes` are allowed:
print(bytes((0, 300))) #causes an out of range error

# Bytes are immutable:
my_array[1] = 41 #causes an error
del my_array[1] #causes an error
```

```
/Users/bianca.schoen
1
b'\x00\x01\x02\x03'
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/
    print(bytes((0, 300))) #causes an out of range error
ValueError: bytes must be in range(0, 256)
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/
    my_array[1] = 41 #causes an error
TypeError: 'bytes' object does not support item assignment
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Coc
    del my_array[1] #causes an error
TypeError: 'bytes' object doesn't support item deletion
```

`bytearray` Exercises

[2]

```
my_array = bytearray((0, 1, 2, 3))
print(my_array[1])

# The bytearray repr:
print(my_array)

# Bytearrays are mutable:
my_array[1] = 41
print(my_array)

print(my_array[1])

# Bytearrays can grow and shrink in size:
del my_array[1]
print(my_array)
my_array.append(42)
print(my_array)
```

```
1
bytearray(b'\x00\x01\x02\x03')
bytearray(b'\x00)\x02\x03')
41
bytearray(b'\x00\x02\x03')
bytearray(b'\x00\x02\x03*')
```

`byteArray` Exercises cont'd

```
# Bytearrays can only hold `bytes`
# (integers in the range 0 <= x <= 255)
my_array[1] = "hello" #causes an error
my_array[1] = 277 #causes an error

# Bytearrays can be converted back into b
# (This will copy the data)
print(type(my_array))
my_array = bytes(my_array)
print(type(my_array))
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/
    my_array[1] = "hello" #causes an error
TypeError: 'str' object cannot be interpreted as an int
```

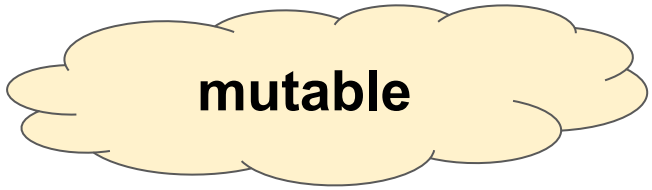
```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Docu
    my_array[1] = 277 #causes an error
ValueError: byte must be in range(0, 25
```

```
<class 'bytearray'>
<class 'bytes'>
```

Records, structs and Data Transfer Objects

- Different to arrays:
  - Fixed number of fields
  - Fields can be of different types
- Several data types in Python to implement these

`dict`



- Dictionary
- Arbitrary number of objects, each with unique key
- Other names: maps, associative arrays
- Efficient for insert, removal and lookup of any element given a key
- No protection again misspelled field names
- Elements can be added and removed at any time

immutable

`tuple` and `namedtuple` and improved `namedtuple`

- tuple: immutable, arbitrary types, access via index position
  - Keep the number of fields low
- Namedtuple:
  - Define a re-usable blueprint for a record
  - Ensures correct field names are used
  - Each object is accessed via a unique identifier, a name, not an index number
  - More memory efficient compared to regular tuples
- Improved namedtuple:
  - Since Python 3.6
  - Updated syntax for defining new record types and type hints support

Often helps to express your intent more clearly, compared to an ad-hoc data type like dict.

Hints are not enforced.

`namedTuple` Exercises

[2]

```
from collections import namedtuple

Car = namedtuple("Car", "color mileage automatic")
my_car = Car("red", 29812.3, False)

# Instances have a nice repr:
print(my_car)

# Accessing fields:
print(my_car.mileage)

# Fields are immtuable:
my_car.mileage = 12 #causes an error

my_car.windshield = "broken" #causes an error
```

```
Car(color='red', mileage=29812.3, automatic=False)
29812.3
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/
    my_car.mileage = 12 #causes an error
AttributeError: can't set attribute
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/
    my_car.windshield = "broken" #causes an error
AttributeError: 'Car' object has no attribute 'windshield'
```

Improved `namedTuple` Exercises

[2]

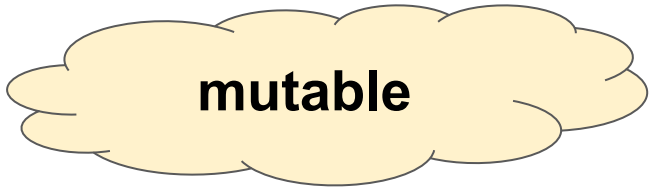
```
from typing import NamedTuple

class Car(NamedTuple):
    color: str
    mileage: float
    automatic: bool

my_car = Car("red", 3812.4, True)

print(my_car)
print(my_car.mileage)

# Fields are immutable:
my_car.mileage = 12 #causes an error
my_car.windshield = "broken" #causes an e

# Type annotations are not enforced without
# a separate type checking tool like mypy:
my_second_car = Car("red", "NOT_A_FLOAT", 99)
print(my_second_car)
```

```
Car(color='red', mileage=3812.4, automatic=True)
3812.4
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_
    my_car.mileage = 12 #causes an error
AttributeError: can't set attribute
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code
    my_car.windshield = "broken" #causes an error
AttributeError: 'Car' object has no attribute 'windshield'
```

`SimpleNamespace`



- Since Python 3.3
- Provides attribute access
- Basically a dictionary that allows attribute access and is easy to print
- Attributes can be added, deleted and changed freely

`simpleNamespace` Exercises

[2]

```
from types import SimpleNamespace
my_car = SimpleNamespace(color="red", mileage=3812.4, automatic=True)
print(my_car)

# Instances support attribute access and are mutable:
my_car.mileage = 12
my_car.windshield = "broken"
del my_car.automatic
print(my_car)
```

```
namespace(automatic=True, color='red', mileage=3812.4)
namespace(color='red', mileage=12, windshield='broken')
```

`struct`

- Serialized c structs
- Import `from struct import Struct`
- Conversion from c structs to Python bytes objects
  - Use case: binary data files coming in via network
  - Rarely used for something that is to be handled purely in Python
  - Data exchange format

`struct` Exercises

[2]

```
from struct import Struct
my_struct = Struct("i?f")
my_data = my_struct.pack(2, False, 41.0) #packed according to a given format
                                         #returns a bytes object


# All you get is a blob of data:
print(my_data)


# Data blobs can be unpacked again:
print(my_struct.unpack(my_data))
```

```
b'\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00$B'
(2, False, 41.0)
```

Write a custom class

- Full control!!!
- Re-usable blueprint
- Fields stored in classes are mutable
- New fields can be added freely
- Great choice if you need to add business logic and behaviours (methods)

`dataclass`

- Available since Python 3.7
- Alternative to defining your own class from scratch
- Lots of useful features out of the box that you don't have to implememt
- Supports type hints
- Import via `from dataclasses import dataclass`
- Uses a decorator

Class Exercise

[2]

```
class Car:
    def __init__(self, color, mileage, automatic):
        self.color = color
        self.mileage = mileage
        self.automatic = automatic

my_first_car = Car("red", 22812.4, True)
my_second_car = Car("grey", 40357.7, False)

# Get the mileage
print(my_second_car.mileage)


# Classes are mutable:
my_second_car.mileage = 12
print(my_second_car.mileage)
my_second_car.windshield = "broken"
print(my_second_car)
# String representation is not very useful
print(my_first_car)
```

```
40357.7
12
<__main__.Car object at 0x7fe8b509adf0>
<__main__.Car object at 0x7fe8b509ad30>
```

`dataClass` Exercise

[2]

```
from dataclasses import dataclass
@dataclass
class Car:
    color: str
    mileage: float
    automatic: bool

my_first_car = Car("red", 3812.4, True)

# Instances have a nice repr:
print(my_first_car)


# Accessing fields:
print(my_first_car.mileage)

# Fields are mutable:
my_first_car.mileage = 12
my_first_car.windshield = "broken"
print(my_first_car.mileage)

# Type annotations are not enforced without
# a separate type checking tool like mypy:
my_second_car = Car("red", "NOT_A_FLOAT", 99)
print(my_second_car)
```

```
Car(color='red', mileage=3812.4, automatic=True)
3812.4
12
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

Sets and Multisets

- Unordered
- Doesn’t allow duplicate elements
- Typically used to quickly test an element for membership in different groups
  - Membership tests are fast, typically O(1)
  - Union, intersection, subset, typically O(n)
- Mutable
- Or use the frozenSet which is immutable
- Mutliset: bag, allows more than one occurance

[3]

`set` and `frozenSet` Example

[2]

```
vowels = {"a", "e", "i", "o", "u"}
print("e" in vowels)


letters = set("bianca")
print(letters.intersection(vowels))

vowels.add("x")
print(vowels)
print(len(vowels))
```

```
True
{'a', 'i'}
{'a', 'u', 'i', 'x', 'o', 'e'}
6
```

```
vowels = frozenset({"a", "e", "i", "o", "u"})
vowels.add("p")
```

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Cod
    vowels.add("p")
AttributeError: 'frozenset' object has no attribute 'add'
```

When do I use what?

Store arbitrary objects, potentially with mixed data types	List or tuple, depending on if you need mutability
You have numeric data and need tight packing	array.array
You have textual data in unicode format	Str, if you need mutability use list of characters
You have a continuous block of bytes	Use bytes or bytearray

When do I use what?

Only a few fields, with names that are easy to remember or superfluous like (x y z) coord	tuple
Need immutable fields	Tuple, namedTuple, improved namedTuple
Need to lock in field types	Improved namedTuples
Keep things simple (plus looks like JSON)	dict
Need full control	Write your own data class
Need to add behaviour	Write your own class or use dataclass decorator
Pack data tightly, serialise to disk or send over network	struct

Summary

- ★ **List**
- ★ **List comprehension**
- ★ **Tuple, namedTuple, improved namedTuple**
- ★ **Dict, struct, object and dataclass**
- ★ **Sets, frozenSets and MultiSet**

References

1. Python 3: Object Oriented Programming, Dusty Phillips, 2nd edition, 2015
2. Real Python: Data Structures, Dan Bader, 26 Aug 2020, https://realpython.com/python-data-structures/, accessed Oct 2020.
3. Time Complexity of Python Datatypes, August 2020, https://wiki.python.org/moin/TimeComplexity, accessed Oct 2020