Data Science

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/DataScience/

42sbxhs

Data Science

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/DataScience/

42sbxhs

Instructor

Assoc. Prof. Dr.
Bora CANBULA

Phone

0 (236) 201 21 08

Email

bora.canbula@cbu.edu.tr

Office Location

Dept. of CENG

Office C233

Office Hours

4 pm – 5 pm, Mondays

Course Overview

Data Science (Teams Code: 42sbxhs)

We are going to try to develop practical data science abilities and programming skills for data science projects in this course. Python is preferred as the programming language for the applications of this course.

Required Text

Introduction to Data Science, Springer, *L. Igual - S. Segui*

Data Science Concepts and Practice, Morgan Kaufmann, *V. Kotu – B. Deshpande*

Course Materials

- Python 3.x (Anaconda is preferred)
- Jupyter Notebook from Anaconda
- Pycharm from JetBrains / Microsoft Visual Studio Code
- PC with a Linux distro or a Linux terminal in Windows 10/11.

Data Science

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/DataScience/

42sbxhs

Course Schedule

Week	Subject	Week	Subject
01	Basic Concepts in Python	08	Midterm Project Presentations – Part 1
02	Introduction to Data Science with Python	09	Midterm Project Presentations – Part 2
03	Data Collections and Preprocessing	10	Advanced Machine Learning Techniques
04	Exploratory Data Analysis (EDA)	11	Model Deployment and Visualization
05	Feature Engineering and Selection	12	Real-time Data and Model Updating
06	Introduction to Machine Learning Models	13	Final Project Presentations – Part 1
07	Model Evaluation and Hyperparameter Tuning	14	Final Project Presentations – Part 2

Data Science

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/DataScience/

42sbxhs

Course Schedule

Week	Subject	Week	Subject
01	Basic Concepts in Python	08	Midterm Project Presentations – Part 1
02	Introduction to Data Science with Python	09	Midterm Project Presentations – Part 2
03	Data Collections and Preprocessing	10	Advanced Machine Learning Techniques
04	Exploratory Data Analysis (EDA)	11	Model Deployment and Visualization
05	Feature Engineering and Selection	12	Real-time Data and Model Updating
06	Introduction to Machine Learning Models	13	Final Project Presentations – Part 1
07	Model Evaluation and Hyperparameter Tuning	14	Final Project Presentations – Part 2

Project Themes

Sports

Economy

Health

Data Science

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/DataScience/

42sbxhs

Sports

Examples

- Predicting Game Outcomes
- Injury Risk Prediction
- Player Market Value Prediction
- Fan Engagement Analysis
- Athlete Performance Comparison

Economy

Examples

- Stock Market Prediction
- Cryptocurrency Price Prediction
- Consumer Spending Analysis
- Predicting Unemployment Rates
- Credit Scoring Model

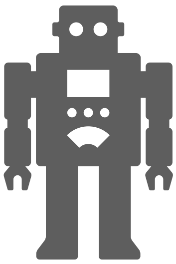
Health

Examples

- Disease Outbreak Prediction
- Personalized Health Recommendations
- Health Risk Prediction
- Hospital Readmission Prediction
- Nutritional Deficiency Prediction

Natural Languages vs. Programming Languages

A language is a tool for expressing and recording thoughts.

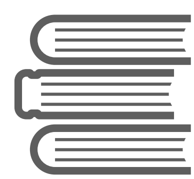


Computers have their own language called **machine** language. Machine languages are created by humans, no computer is currently capable of creating a new language. A complete set of known commands is called an instruction list (IL).

The difference is that human languages developed naturally. They are still evolving, new words are created every day as old words disappear. These languages are called **natural** languages.

Elements of a Language

- **Alphabet** is a set of symbols to build words of a certain language.
- **Lexis** is a set of words the language offers its users.
- **Syntax** is a set of rules used to determine if a certain string of words forms a valid sentence.
- **Semantics** is a set of rules determining if a certain phrase makes sense.



Machine Language vs. High-Level Language

The IL is the alphabet of a machine language. It's the computer's mother tongue.

High-level programming language enables humans to write their programs and computers to execute the programs. It is much more complex than those offered by ILs.

A program written in a high-level programming language is called a **source code**. Similarly, the file containing the source code is called the **source file**.

Compilation vs. Interpretation

There are two different ways of transforming a program from a high-level programming language into machine language:

Compilation: The source code is translated once by getting a file containing the machine code.

Interpretation: The source code is interpreted every time it is intended to be executed.

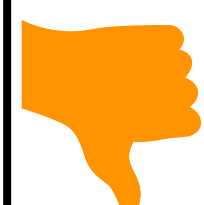
Compilation

- The execution of the translated code is usually faster.
- Only the user has to have the compiler. The end user may use the code without it.
- The translated code is stored using machine language. Your code are likely to remain your secret.

- The compilation itself may be a very time-consuming process
- You have to have as many compilers as hardware platforms you want your code to be run on.

Interpretation

- You can run the code as soon as you complete it, there are no additional phases of translation.
- The code is stored using programming language, not machine language. You don't compile your code for each different architecture.



- Don't expect interpretation to ramp up your code to high speed
- Both you and the end user have the interpreter to run your code.

What is Python?

Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.

Python was created by Guido van Rossum. The name of the Python programming language comes from an old BBC television comedy sketch series called Monty Python's Flying Circus.

Python Goals

- an **easy and intuitive** language just as powerful as those of the major competitors
- **open source**, so anyone can contribute to its development
- code that is as **understandable** as plain English
- **suitable for everyday tasks**, allowing for short development times

Why Python?

- easy to learn
- easy to teach
- easy to use
- easy to understand
- easy to obtain, install and deploy

Why not Python?

- low-level programming
- applications for mobile devices

Python Implementations

An implementation refers to a program or environment, which provides support for the execution of programs written in the Python language.

- **CPython** is the traditional implementation of Python and it's most often called just "Python".
- **Cython** is a solution which translate Python code into "C" to make it run much faster than pure Python.
- **Jython** is an implementation follows only Python 2, not Python 3, written in Java.
- **PyPy** represents a Python environment written in Python-like language named RPython (Restricted Python), which is actually a subset of Python.
- **MicroPython** is an implementation of Python 3 that is optimized to run on microcontrollers.

Start Coding with Python

- **Editor** will support you in writing the code. The Python 3 standard installation contains a very simple application named IDLE (Integrated Development and Learning Environment).
- **Console** is a terminal in which you can launch your code.
- **Debugger** is a tool, which launches your code step-by-step to allow you to inspect it.

first.py

```
print("Python is the best!")
```

https://forms.offi[illegible]r/3qrM5Gj66X

https:/[illegible]s.office.com/r/GNNHg5B4c7

in-class quizzes

Function Name
A function can cause some effect or evaluate a value, or both.

Where do functions come from?
- From Python itself
- From modules
- From your code

first.py

```python
print("Python is the best!")
```

Argument
- Positional arguments
- Keyword arguments

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Literals

A literal is data whose values are determined by the literal itself.
Literals are used to encode data and put them into code.

literals.py

```
print("7")
print(7)
print(7.0)
print(7j)
print(True)
print(0b10)
print(0o10)
print(0x10)
print(7.4e3)
```

- String
- Integer
- Float
- Complex
- Boolean
- Binary
- Octal
- Hexadecimal
- Scientific Notation

Basic Operators

An operator is a symbol of the programming language, which is able to operate on the values.

Multiplication

```
print(2 * 3)      Integer
print(2 * 3.0)    Float
print(2.0 * 3)    Float
print(2.0 * 3.0)  Float
```

Division

```
print(6 / 3)      Float
print(6 / 3.0)    Float
print(6.0 / 3)    Float
print(6.0 / 3.0)  Float
```

Exponentiation

```
print(2**3)       Integer
print(2**3.0)     Float
print(2.0**3)     Float
print(2.0**3.0)   Float
```

Floor Division

```
print(6 // 3)     Integer
print(6 // 3.0)   Float
print(6.0 // 3)   Float
print(6.0 // 3.0) Float
```

Modulo

```
print(6 % 3)      Integer
print(6 % 3.0)    Float
print(6.0 % 3)    Float
print(6.0 % 3.0)  Float
```

Addition

```
print(-8 + 4)     Integer
print(-4.0 + 8)   Float
```

Operator Priorities

An operator is a symbol of the programming language, which is able to operate on the values.

priorities.py

```
print(9 % 6 % 2)
print(2**2**3)
print(2 * 3 % 5)
print(-3 * 2)
print(-2 * 3)
print(-(2 * 3))
```

- ** (right-sided binding)
- + (unary)
- - (unary)
- *
- /
- //
- % (left-sided binding)
- + (binary)
- - (binary)

Variables

Variables are symbols for memory addresses.

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions			
A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	**F**	**M**	
ascii()	filter()	map()	**S**
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *x* is not a Python int object, it has to define an __index__() method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

id(*object*)

Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

https://docs.python.org/3/library/functions.html

Identifier Names

For variables, functions, classes etc. we use identifier names. We must obey some rules and we should follow some naming conventions.

- Names are case sensitive.
- Names can be a combination of letters, digits, and underscore.
- Names can only start with a letter or underscore, can not start with a digit.
- Keywords can not be used as a name.

`keyword` — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword or soft keyword.

`keyword.`**`iskeyword`**`(s)`

Return `True` if *s* is a Python keyword.

`keyword.`**`kwlist`**

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.

`keyword.`**`issoftkeyword`**`(s)`

Return `True` if *s* is a Python soft keyword.

New in version 3.9.

`keyword.`**`softkwlist`**

Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.

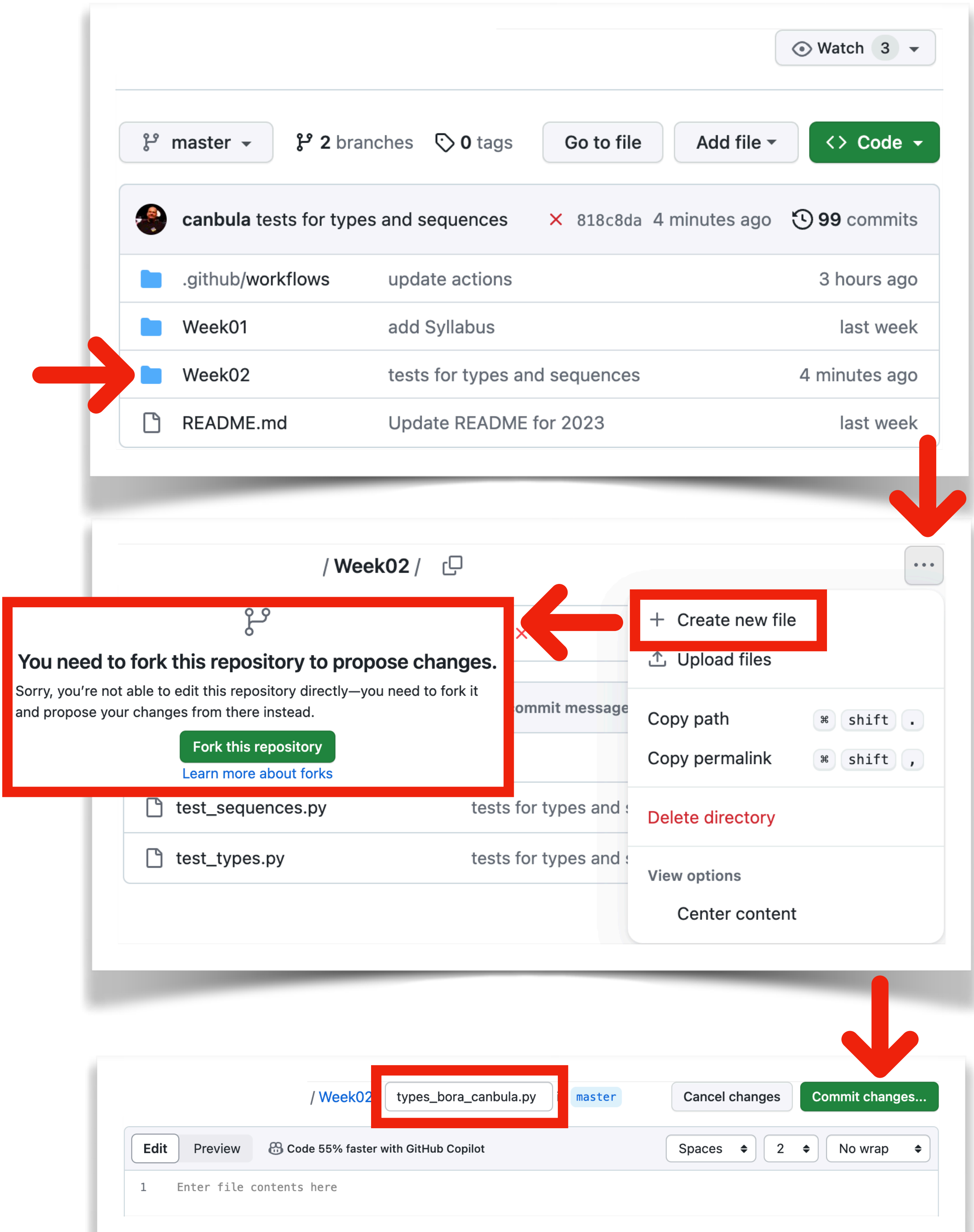
New in version 3.9.

https://forms.offic /VBi2yJZiX5

in-class quizzes

https:/ office.com/r/RRCyEr8RE8

Your First Homework



- An integer with the name: my_int
- A float with the name: my_float
- A boolean with the name: my_bool
- A complex with the name: my_complex

Equality & Identity & Comparison

The operators `is` and `is not` test for an object's identity: `x is y` is true if and only if *x* and *y* are the same object. An Object's identity is determined using the `id()` function. `x is not y` yields the inverse truth value. [4]

Equality

```
a = 4
print(a == 4) ✓
print(id(a) == id(4)) ✓
print(a is 4) ✓
print(id(a) is id(4)) ✗
```

Left- or Right-sided?

```
x, y, z = 0, 1, 2
print(x == y == z) ✗
print(x == (y == z)) ✓
print((x == y) == z) ✗
```

Inequality

```
print(1 != 2) ✓
print(not 1 == 2) ✓
```

Updated Priority Table

Operator	Type
+, -	unary
**	binary
*, /, //, %	binary
+, -	binary
<, <=, >, >=	binary
!=, ==	binary

Comparison

```
print(1 < 2) ✓
print(1 <= 2) ✓
print(1 > 2) ✗
print(1 >= 2) ✗
```

Chaining

```
print(1 < 2 < 3) ✓
print(1 < 2 > 3) ✗
print(1 < 2 >= 3) ✗
print(1 < 2 <= 3) ✓
```

QUESTION

Using one of the comparison operators in Python, write a simple two-line program that takes the parameter **n** as input, which is an integer, prints **False** if **n** is less than **100**, and **True** if **n** is greater than or equal to **100**.

```
n = int(input())
print(n >= 100)
```

Conditional Execution

if statement

```
if n >= 100:
    print("The number is greater than or equal to 100.")
elif n < 0:
    print("The number is negative.")
else:
    print("The number is less than 100.")
```

Ternary Operator

```
msg = "The number is greater than or equal to 100." if n >= 100 else "The number is less than 100."
print(msg)
```

Loops

QUESTION

- The program generates a random number between 1 and 10.
- The user is asked to guess the number.
- The user is given feedback if the guess is too low or too high.
- The user is asked to guess again until the correct number is guessed.

```
r = random.randint(1, 10)
answer = False
while not answer:
    n = int(input("Enter a number: "))
    if n == r:
        print("You guessed it right!")
        answer = True
    elif n < r:
        print("Try a higher number.")
    else:
        print("Try a lower number.")
```

QUESTION

- The user is asked to enter a number.
- The program prints the numbers from 0 to n-1.

```
n = int(input("Enter a number: "))
for i in range(n):
    print(i, end=" ")
```

break and continue

```
n = int(input("Enter a number: "))
for i in range(10):
    if i < n:
        print("The number is not found:", i)
        continue
    if i == n:
        print("The number is found:", i)
        break
```

```
r = random.randint(1, 10)
while True:
    n = int(input("Enter a number: "))
    if n == r:
        print("You guessed it right!")
        break
    elif n < r:
        print("Try a higher number.")
    else:
        print("Try a lower number.")
```

class **range**(*stop*)
class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

start
The value of the *start* parameter (or `0` if the parameter was not supplied)

stop
The value of the *stop* parameter

step
The value of the *step* parameter (or `1` if the parameter was not supplied)

The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Can we use while/for with else?

LISTS IN PYTHON:

Ordered and mutable sequence of values indexed by integers

Initializing

```
a_list = [] ## empty
a_list = list() ## empty
a_list = [3, 4, 5, 6, 7] ## filled
```

Finding the index of an item

```
a_list.index(5) ## 2 (the first occurence)
```

Accessing the items

```
a_list[0] ## 3
a_list[1] ## 4
a_list[-1] ## 7
a_list[-2] ## 6
a_list[2:] ## [5, 6, 7]
a_list[:2] ## [3, 4]
a_list[1:4] ## [4, 5, 6]
a_list[0:4:2] ## [3, 5]
a_list[4:1:-1] ## [7, 6, 5]
```

Adding a new item

```
a_list.append(9) ## [3, 4, 5, 6, 7, 9]
a_list.insert(2, 8) ## [3, 4, 8, 5, 6, 7, 9]
```

Update an item

```
a_list[2] = 1 ## [3, 4, 1, 5, 6, 7, 9]
```

Remove the list or just an item

```
a_list.pop() ## last item
a_list.pop(2) ## with index
del a_list[2] ## with index
a_list.remove(5) ## first occurence of 5
a_list.clear() ## returns an empty list
del a_list ## removes the list completely
```

```
a_list[4:1:-1] ## [7, 6, 5]
```

Adding a new item

```
a_list.append(9) ## [3, 4, 5, 6, 7, 9]
a_list.insert(2, 8) ## [3, 4, 8, 5, 6, 7, 9]
```

Update an item

```
a_list[2] = 1 ## [3, 4, 1, 5, 6, 7, 9]
```

Remove the list or just an item

```
a_list.pop() ## last item
a_list.pop(2) ## with index
del a_list[2] ## with index
a_list.remove(5) ## first occurence of 5
a_list.clear() ## returns an empty list
del a_list ## removes the list completely
```

Extend a list with another list

```
list_1 = [4, 2]
list_2 = [1, 3]
list_1.extend(list_2) ## [4, 2, 1, 3]
```

Reversing and sorting

```
list_1.reverse() ## [3, 1, 2, 4]
list_1.sort() ## [1, 2, 3, 4]
```

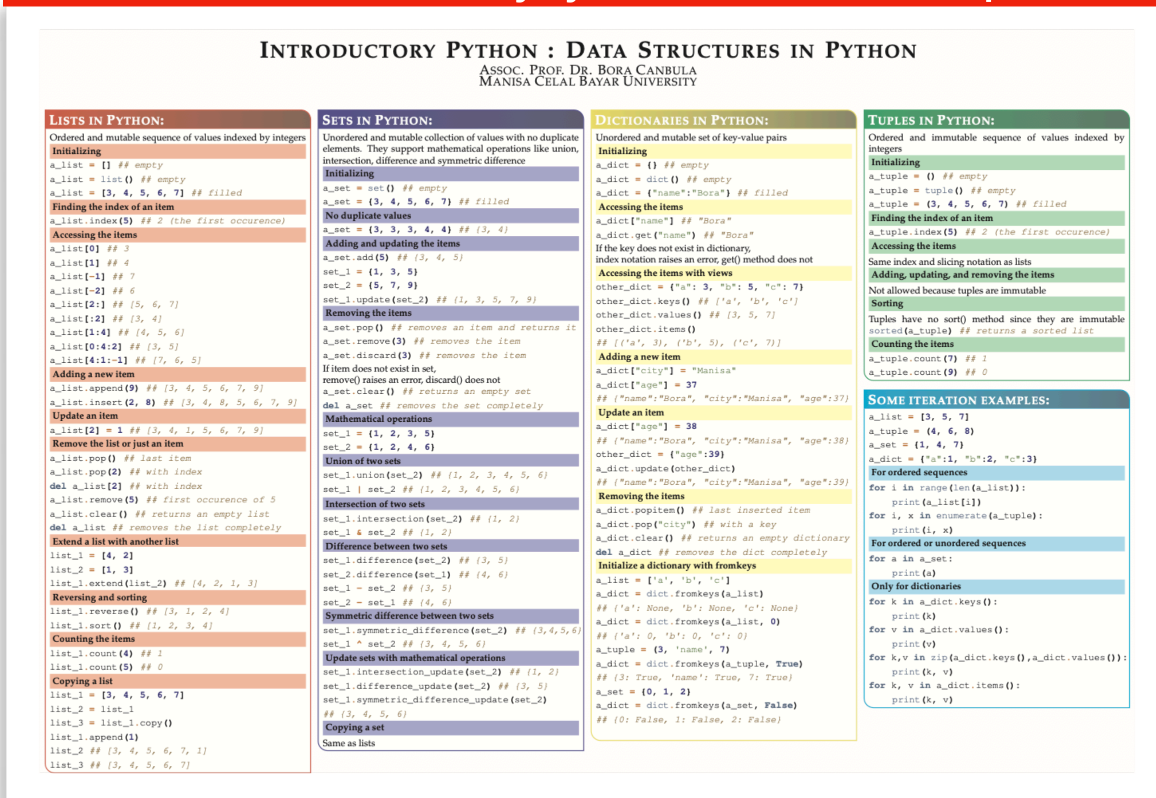
Counting the items

```
list_1.count(4) ## 1
list_1.count(5) ## 0
```

Copying a list

```
list_1 = [3, 4, 5, 6, 7]
list_2 = list_1
list_3 = list_1.copy()
list_1.append(1)
list_2 ## [3, 4, 5, 6, 7, 1]
list_3 ## [3, 4, 5, 6, 7]
```

Week03/IntroductoryPythonDataStructures.pdf



INTRODUCTORY PYTHON : DATA STRUCTURES IN PYTHON

ASSOC. PROF. DR. BORA CANBULA
MANISA CELAL BAYAR UNIVERSITY

LISTS IN PYTHON:

Ordered and mutable sequence of values indexed by integers

Initializing

```
a_list = [] ## empty
a_list = list() ## empty
a_list = [3, 4, 5, 6, 7] ## filled
```

Finding the index of an item

```
a_list.index(5) ## 2 (the first occurence)
```

Accessing the items

```
a_list[0] ## 3
a_list[1] ## 4
a_list[-1] ## 7
a_list[-2] ## 6
a_list[2:] ## [5, 6, 7]
a_list[:2] ## [3, 4]
a_list[1:4] ## [4, 5, 6]
a_list[0:4:2] ## [3, 5]
a_list[4:1:-1] ## [7, 6, 5]
```

Adding a new item

```
a_list.append(9) ## [3, 4, 5, 6, 7, 9]
a_list.insert(2, 8) ## [3, 4, 8, 5, 6, 7, 9]
```

Update an item

```
a_list[2] = 1 ## [3, 4, 1, 5, 6, 7, 9]
```

Remove the list or just an item

```
a_list.pop() ## last item
a_list.pop(2) ## with index
del a_list[2] ## with index
a_list.remove(5) ## first occurence of 5
a_list.clear() ## returns an empty list
del a_list ## removes the list completely
```

Extend a list with another list

```
list_1 = [4, 2]
list_2 = [1, 3]
list_1.extend(list_2) ## [4, 2, 1, 3]
```

Reversing and sorting

```
list_1.reverse() ## [3, 1, 2, 4]
list_1.sort() ## [1, 2, 3, 4]
```

Counting the items

```
list_1.count(4) ## 1
list_1.count(5) ## 0
```

Copying a list

```
list_1 = [3, 4, 5, 6, 7]
list_2 = list_1
list_3 = list_1.copy()
list_1.append(1)
list_2 ## [3, 4, 5, 6, 7, 1]
list_3 ## [3, 4, 5, 6, 7]
```

SETS IN PYTHON:

Unordered and mutable collection of values with no duplicate elements. They support mathematical operations like union, intersection, difference and symmetric difference

Initializing

```
a_set = set() ## empty
a_set = {3, 4, 5, 6, 7} ## filled
```

No duplicate values

```
a_set = {3, 3, 3, 4, 4} ## {3, 4}
```

Adding and updating the items

```
a_set.add(5) ## {3, 4, 5}
set_1 = {1, 3, 5}
set_2 = {5, 7, 9}
set_1.update(set_2) ## {1, 3, 5, 7, 9}
```

Removing the items

```
a_set.pop() ## removes an item and returns it
a_set.remove(3) ## removes the item
a_set.discard(3) ## removes the item
```

If item does not exist in set,
remove() raises an error, discard() does not

```
a_set.clear() ## returns an empty set
del a_set ## removes the set completely
```

Mathematical operations

```
set_1 = {1, 2, 3, 5}
set_2 = {1, 2, 4, 6}
```

Union of two sets

```
set_1.union(set_2) ## {1, 2, 3, 4, 5, 6}
set_1 | set_2 ## {1, 2, 3, 4, 5, 6}
```

Intersection of two sets

```
set_1.intersection(set_2) ## {1, 2}
set_1 & set_2 ## {1, 2}
```

Difference between two sets

```
set_1.difference(set_2) ## {3, 5}
set_2.difference(set_1) ## {4, 6}
set_1 - set_2 ## {3, 5}
set_2 - set_1 ## {4, 6}
```

Symmetric difference between two sets

```
set_1.symmetric_difference(set_2) ## {3,4,5,6}
set_1 ^ set_2 ## {3, 4, 5, 6}
```

Update sets with mathematical operations

```
set_1.intersection_update(set_2) ## {1, 2}
set_1.difference_update(set_2) ## {3, 5}
set_1.symmetric_difference_update(set_2)
## {3, 4, 5, 6}
```

Copying a set

Same as lists

DICTIONARIES IN PYTHON:

Unordered and mutable set of key-value pairs

Initializing

```
a_dict = {} ## empty
a_dict = dict() ## empty
a_dict = {"name":"Bora"} ## filled
```

Accessing the items

```
a_dict["name"] ## "Bora"
a_dict.get("name") ## "Bora"
```

If the key does not exist in dictionary, index notation raises an error, get() method does not

Accessing the items with views

```
other_dict = {"a": 3, "b": 5, "c": 7}
other_dict.keys() ## ['a', 'b', 'c']
other_dict.values() ## [3, 5, 7]
other_dict.items()
## [('a', 3), ('b', 5), ('c', 7)]
```

Adding a new item

```
a_dict["city"] = "Manisa"
a_dict["age"] = 37
## {"name":"Bora", "city":"Manisa", "age":37}
```

Update an item

```
a_dict["age"] = 38
## {"name":"Bora", "city":"Manisa", "age":38}
other_dict = {"age":39}
a_dict.update(other_dict)
## {"name":"Bora", "city":"Manisa", "age":39}
```

Removing the items

```
a_dict.popitem() ## last inserted item
a_dict.pop("city") ## with a key
a_dict.clear() ## returns an empty dictionary
del a_dict ## removes the dict completely
```

Initialize a dictionary with fromkeys

```
a_list = ['a', 'b', 'c']
a_dict = dict.fromkeys(a_list)
## {'a': None, 'b': None, 'c': None}
a_dict = dict.fromkeys(a_list, 0)
## {'a': 0, 'b': 0, 'c': 0}
a_tuple = (3, 'name', 7)
a_dict = dict.fromkeys(a_tuple, True)
## {3: True, 'name': True, 7: True}
a_set = {0, 1, 2}
a_dict = dict.fromkeys(a_set, False)
## {0: False, 1: False, 2: False}
```

TUPLES IN PYTHON:

Ordered and immutable sequence of values indexed by integers

Initializing

```
a_tuple = () ## empty
a_tuple = tuple() ## empty
a_tuple = (3, 4, 5, 6, 7) ## filled
```

Finding the index of an item

```
a_tuple.index(5) ## 2 (the first occurence)
```

Accessing the items

Same index and slicing notation as lists

Adding, updating, and removing the items

Not allowed because tuples are immutable

Sorting

Tuples have no sort() method since they are immutable

```
sorted(a_tuple) ## returns a sorted list
```

Counting the items

```
a_tuple.count(7) ## 1
a_tuple.count(9) ## 0
```

SOME ITERATION EXAMPLES:

```
a_list = [3, 5, 7]
a_tuple = (4, 6, 8)
a_set = {1, 4, 7}
a_dict = {"a":1, "b":2, "c":3}
```

For ordered sequences

```
for i in range(len(a_list)):
    print(a_list[i])
for i, x in enumerate(a_tuple):
    print(i, x)
```

For ordered or unordered sequences

```
for a in a_set:
    print(a)
```

Only for dictionaries

```
for k in a_dict.keys():
    print(k)
for v in a_dict.values():
    print(v)
for k,v in zip(a_dict.keys(),a_dict.values()):
    print(k, v)
for k, v in a_dict.items():
    print(k, v)
```

https://forms.offic /r/WVGNuHabiV

https:// office.com/r/cpBQGv3NJ0

in-class quizzes

Week03/sequences_first_last.py

```
def remove_duplicates(seq: list) -> list:
    """
    This function removes duplicates from a list.
    """
    return ...

def list_counts(seq: list) -> dict:
    """
    This function counts the number of
    occurrences of each item in a list.
    """
    return ...

def reverse_dict(d: dict) -> dict:
    """
    This function reverses the keys
    and values of a dictionary.
    """
    return ...
```

Iterables - Sequences - Iterators

An **iterable** is any object that can be looped over. It represents a collection of elements that can be accessed one by one.

An object is considered iterable if:

- It implements the `__iter__()` method which returns an iterator, or
- It defines the `__getitem__()` method that can fetch items using integer indices starting from zero.

A **sequence** is a subtype of iterables. It's an ordered collection of elements that can be indexed by numbers.

- **Ordered:** Elements in a sequence have a specific order.
- **Indexable:** You can get any item using an index `my_sequence[5]`.
- **Slicable:** Supports slicing to get some of items `my_sequence[2:5]`.

An **iterator** is an object that produces items (one at a time) from its associated iterable.

- **Stateful:** An iterator remembers its state between calls. Once an element is consumed, it can't be accessed again without reinitializing the iterator.
- **Lazy Evaluation:** Items are not produced from the source iterable until the iterator's `__next__()` method is called.
- Iterators raise a StopIteration exception when there are no more items to return.
- An iterator's `__iter__()` method returns the iterator object itself.
- While all iterables must be able to produce an iterator (with `__iter__()` method), not all iterators are directly iterable without using a loop.

Numpy Arrays

Numerical Python (**NumPy**)is a powerful library for numerical computing. Its key feature is multi dimensional arrays (**ndarrays**).

Traditional Python Lists

- **Dynamically Typed:** Lists can store elements of mixed types in a single list.
- **Resizable:** Lists can be resized by appending or removing elements.
- **General-purpose:** Lists are general-purpose containers for items of any type.
- **Memory:** Lists have a larger memory overhead because of their general-purpose nature and dynamic typing.
- **Performance:** Basic operations on lists may not be as fast as those on NumPy arrays because they aren't optimized for numerical operations.

NumPy Arrays

- **Typed:** All elements in a NumPy array are of the same type.
- **Size:** The size of a NumPy array is fixed upon creation. However, one can create a new array with a different size, but resizing in-place (like appending in lists) isn't directly supported.
- **Efficiency:** NumPy arrays are memory-efficient as they store elements in contiguous blocks of memory.
- **Performance:** Operations on NumPy arrays are typically faster than lists, especially for numerical tasks, due to optimized C and Fortran extensions.
- **Vectorized Operations:** Supports operations that apply to the entire array without the need for explicit loops (e.g., adding two arrays element-wise).
- **Broadcasting:** Advanced feature allowing operations on arrays of different shapes.
- **Extensive Functionality:** Beyond just array storage, NumPy provides a vast range of mathematical, logical, shape manipulation, and other operations.
- **Interoperability:** Can interface with C, C++, and Fortran code.

Homework

Submit your work to GitHub

Week04/arrays_firstname_lastname.py

Function Description

replace_center_with_minus_one(d, n, m)

This function creates an `n-by-n` numpy array populated with random integers that have up to `d` digits. It then replaces the central `m-by-m` part of this array with `-1`.

Parameters

- **d**: Number of digits for the random integers.
- **n**: Size of the main array.
- **m**: Size of the central array that will be replaced with `-1`.

Returns

- A modified numpy array with its center replaced with `-1`.

Exceptions

- **ValueError**: This exception is raised in the following scenarios:
  - If `m > n`
  - If `d <= 0`
  - If `n < 0`
  - If `m < 0`