# Data Types

Prof.Dr. Bahadır AKTUĞ
Machine Learning with Python

*Compiled from sources given in the references.

# Statically vs. Dynamically Typed Languages

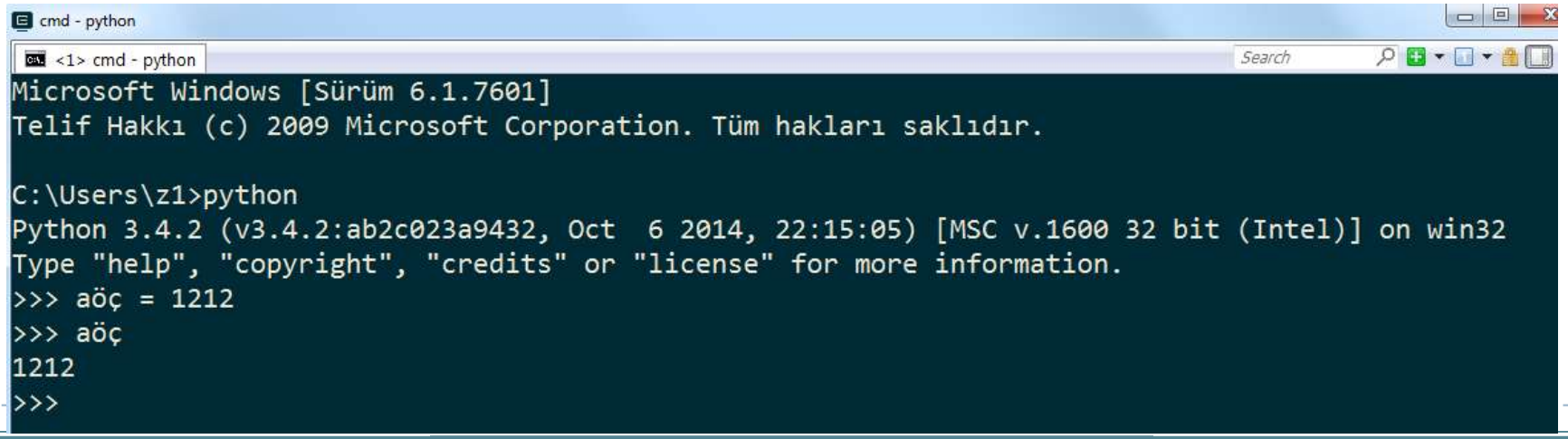
- In statically typed languages, the variables have to be defined before they are used (C/C++/Pascal etc.).
- In statically typed languages, a variable can only have one type that cannot be changed during the program execution.
- In a dynamically typed languages, the variables do not have to be defined before they are assigned.
- In a dynamically typed language, the variables can change their type during the runtime.
- For instance, while variable is an integer at the beginning of a program and then it can be string at the end.
- Python is a dynamically typed programming language.

# Strongly vs. Weakly Typed Languages

- In strongly typed languages, the operators take the type of each operand into account and a check called "type safety" is applied (C/C++/Pascal etc.).
  - a = "Python"
  - a=1457
  - a = input()
  - print(int(a))
- In a strongly typed language, you cannot add a number to a string or vice versa.
- In a weakly typed language, the usage of the different data types are flexible (Perl, Javascript).
- Python is a strongly typed programming language.

# Python Variable Names

- The naming convention with Python 3 has been made quite flexible.
- The variable naming restrictions in Python 3 can be summarized as below:
  - The first character of a variable name must be either a letter (lowercase or uppercase) or "_"
  - The letter could be Unicode
  - Any letter or number can follow after the first character.



# Python Variable Names

- Python is a "case sensitive" language. This also applies to variable as well as commands, functions etc.
- The variable names cannot be chosen from the reserved word list below (they are python commands!)
  - *and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield*

# Numbers in Python

**Integers**

- Decimals (numbers on base 10)
- Octals (numbers in base 8): (they must have "0" and "o")

  >>> a = 0o20

  >>> print(a)

  >>> 16
- Hexadecimals (numbers on base 16): (they must have "0" and "x/X")

  >>> a = 0x10

  >>> print(a)

  >>> 16
- Binaries (numbers on base 2): (they must have "0" and "b/B")
- a = 0b110

  >>> print(a)

  >>> 6

# Conversion to a different base

- Decimal numbers can be converted to other bases:
- From decimal to octal (base 8):

  >>> a = 16

  >>> print(oct(a))

  >>> '0o20'  (note that it is converted as a string)
- From decimal to base 16:
- >>> a = 16

  >>> print(hex(a))

  >>> '0x10' (note that it is converted as a string)
- From decimal to base 2:
- >>> a = 16

  >>> print(bin(a))

  >>> '0b10000' (note that it is converted as a string)

# Numbers in Python

## Integers

- There is no limit for integers:

```
>>> x = 787366098712738903245678234782358292837498729182728
>>> x * x * x
488123970070638215986770162105731315538827586091948617997871122950228891123960901918308618286311523282239313708275589787123005317148968569797875581092352
```

## Floating Numbers

```
>>> a = 14.56
>>> a = 2.4583e-8
```

## Complex Numbers

- Complex numbers can directly be used in Python.

```
>>> a = 3 – 5j
>>> b = 4 +7j
>>> a+b
```

# String type

- There is a need for "string" type to express a sequence of characters (letters, alphanumeric, even numbers, special characters etc).
- ASCII coding allows defining 256 ($2^8$) different characters.
- However, there are far more letters and symbols than can be accommodated by ASCII. Thus, Unicode standard was established.
- Unicode uses a 4-byte representation instead of ASCII's 1 byte representation of characters.
- 4-byte representation of Unicode allows $(2^8)^4$ > 4 million different characters.
- Since Unicode's 4 byte representation (character mapping) allocates 4-bytes even for characters where 1 byte is sufficient, different Unicode Codings were developed (UF-8, UTF-16 ve UTF-32)

# String type in Python

- The string are defined as Unicode in Python without any coding.
- The string types can be defined with a single or double quote:

```
>>> a = 'EEE105'
>>> a = "EEE105"
```

- If the character sequence to be assigned to a string variable already contains a single/double quote, a backslash (\) should be used before it. If the string variable is defined with a single quote, the quote inside could de double or vice versa.

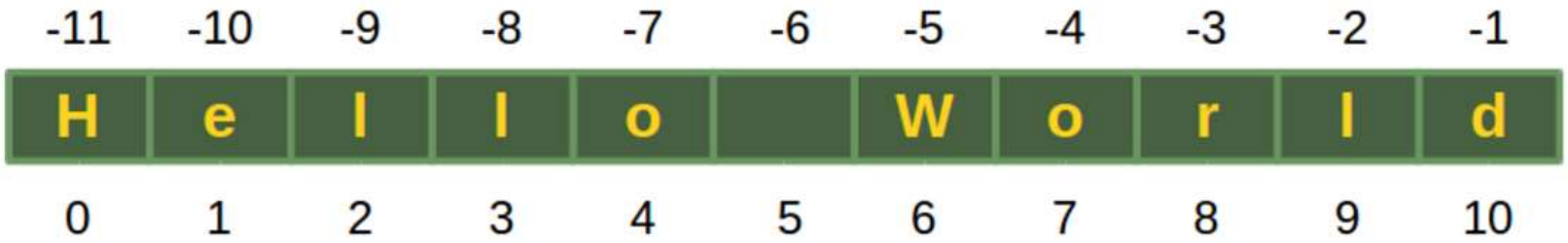
```
>>> a = 'EEE105\'s content'
>>> a = "EEE105\"s content"
```

- There is also a triple quote in Python which is used to define a multiline comment.

# String type in Python

- A single character of a string variable in Python can be directly accesed with indexing.

```
>>> s = 'Hello World'
>>> s[0]
>>> 'H'
```




- The last characters can be accessed by using either of the following methods:

```
>>> s[len(s)-1]
>>> 'd'
>>> s[-1]
>>> 'd'
```

# String type in Python

## Concatenation:

- String concatenation is done by using operator "+":

```
>>> a = 'EEE105'
>>> b = " Computer Programming I"
>>> a+b
>>> 'EEE105 Computer Programming'
```

## Repetition:

- A repetition of string is done using operator "*":

```
>>> a = 'AB'
>>> 3*a
>>> 'ABABAB'
```

# String type in Python

## Indexing:

- Indexing in Python is done through operator "[]".
- Python allows for negative indexing.

```
>>> a = 'AB'
>>> a[1]
>>> 'B'
>>> a[0]
>>> 'A'
>>> a[-1]
>>> 'B'
>>> a[-6]
>>> 'A'
>>> a[-7]
>>> Hata mesajı
```

**Indexing starts from 0!*

**After reaching the start of the variable it does not go back!*

# String type in Python

## Slicing:

- Slicing in Python is done through operators "[:]"
- The start/end indices take place on the left and right side of ":"

```
>>> a = 'Ankara'
>>> a[3:5]
>>> 'ar'
```

- The start/end indices can be left blank. In this case, it means from the start/to the end:

```
>>> a[:4]
>>> 'Anka'
>>> a[4:]
>>> 'ra'
```

**Please note that slicing starts from index 0 just like indexing and second index is not included!*

# String type in Python

## Size & Length:

- The find the length of a string, len() function is used.
- "len" function gives the number of characters.
- "space" counts.
- To access the last character in a string variable a, the indexing a[len(a)-1] can be used.

```
>>> a = 'Ankara'
>>> len(a)
>>> 6
>>> a = 'Ankara İstanbul'
>>> len(a)
>>> 15
```

# Mutable and Immutable Variables

- Mutable and Immutable variables are closely related to the concepts of "call by value" and "call by reference" which are also examined in the chapter about functions.
- In short, the string data type in Python is an immutable type. This means that the letters of a string cannot be modified by usual assignment.

```
>>> a = 'Ankara'
>>> a[0] = 'O'
error message .....
```

**it tries to change the string to "Onkara"*

# How is a string variable kept in the memory?

- Almost anything in Python is an object and is kept in at a specific memory address. The content (value) of variables can be compared with the operator "==". But to check whether they are point at the same memory address, "is" operator is used:

```
>>> a = 'Ankara'; b = "Ankara"
>>> a == b
>>> True
>>> a is b
>>> True
>>> a = 'Med-Cezir'
>>> b = "Med-Cezir"
>>> a == b
>>> True
>>> a is b
>>> False
```

**They are pointing at the same object (the same memory address). Their contents are the same*

**They are not pointing at the same object (the same memory address). Their contents are the same*

# String Variables in Python

## Escape Sequences:

- String variables can contain special characters.
- They must have operator "\" to discriminate them against the usual characters.

| Escape Sequence | Meaning Notes |
|---|---|
| \newline | Ignored |
| \\ | Backslash (\) |
| \' | Single quote (') |
| \" | Double quote (") |
| \a | ASCII Bell (BEL) |
| \b | ASCII Backspace (BS) |
| \f | ASCII Formfeed (FF) |
| \n | ASCII Linefeed (LF) |
| \N{name} | Character named name in the Unicode database (Unicode only) |
| \r | ASCII Carriage Return (CR) |
| \t | ASCII Horizontal Tab (TAB) |
| \uxxxx | Character with 16-bit hex value xxxx (Unicode only) |
| \Uxxxxxxxx | Character with 32-bit hex value xxxxxxxx (Unicode only) |
| \v | ASCII Vertical Tab (VT) |
| \ooo | Character with octal value ooo |
| \xhh | Character with hex value hh |

# Variable Assignment

- The assignment operator is "=" as is in many programming languages.
- Python is a dynamically typed language. The content of the variable (its value) determines the data type.
- The very same variable can have different data types within the same code block.
- On the other hand, Python is a strongly typed language. Once the type is determined depending on the content, the operators should be compatible.

```
>>> a = "Gölbaşı"
>>> a = 27e12
>>> a = 1451 * 2321
```

# Variable Assignment

- When we take into account that all the variables in Python are actually objects, caution should be exercised while assigning variables to one another.
- When we assign a value to a variable, a chunk of memory is allocated and an address of memory is assigned.
- When we assign variables to each other, only the memory address is assigned not their values.
- Unless deliberately done, such phenomenon could have disastrous results. When the content of the assigned variable is modified, it also effects the first variable content.
- Python handles such a situation by assigning a new address during each value assignment.

```
>>> a = [2,4,5]
>>> b = a
>>> b[0] = 1
>>> a
>>> [1,4,5]
```

# Sequential Data Types

# Sequential Data Types

- Sequential data types are needed in programming at any scale.
- Python provides six sequential data types:
  - strings
  - byte sequence
  - byte arrays
  - list
  - tuple
  - range object
- While these data seem to be quite different at first sight, they have one important common feature: they hold data sequentially.

# Sequential Data Types

- The elements of a sequential data type can be accessed with indexing.
- Remember the indexing we use to access characters in a string type variable:

```
>>> s = "Programming with Python"
>>> print(s[0], s[17])
PP
```

- Accessing the elements of a list with indexing:

```
>>> l = ["Ankara", "İstanbul", "İzmir", "Adana"]
>>> print(l[1], l[2])
İstanbul İzmir
```

# Sequential Data Types

- There are also functions defined for sequential data types. In Python, such functions are common to all sequential data types (string, list, tuple etc.).
- For instance, the length of a sequential data type can retreived by using "len()" function:

```
>>> s = "Programming with Python"
>>> l = ["Ankara", "İstanbul", "İzmir", "Adana"]
>>> print(len(s),len(l))
23 4
```

# Lists

- In general, "lists" can be considered similar to the arrays in C, C++, Java and Matlab.
- However, "lists" in Python are much more powerful and flexible with respect to the "arrays" in classical programming languages.
- For one thing, the elements of a "list" does not have to be of the same data type (integer, string, float etc.).
- Lists can be expanded/shrunk during runtime. In static arrays, the dimension is constant during run time.
- The lists in Python are an array of sequential objects. Those objects could be any data type including other lists.

# Lists

- Some feature of lists in Python:
  - The elements take place sequentially
  - The elements could be of any data type
  - The access to the elements of a list is done through indexing
  - Lists, lists including other lists (any nested object) could the elements of a list
  - The dimension is not constant
  - Lists are of mutable data type

# Lists

## Some examples of lists in Python:

| Definition | Description |
|---|---|
| [] | empty list |
| [1,1,2,3,5,8] | a list of integers |
| [42, "JFM212", 3.1415] | a list of various data types |
| ["Ankara", "Adana", "Bursa", "İzmir", "Gaziantep", "Antalya","Konya", "Samsun"] | a list of strings |
| [["Ankara","Konya", 7556900], ["New York","Londra",2193031], ["Antalya", "Samsun", 123466]] | a list containing lists as elements |
| ["İller", ["ilçeler", ["beldeler", ["köyler", "mezralar", 1021]]]] | a nested list |

# Lists

- Access to the elements and sub-elements of a list:
  - Indexing is used to access to the elements.
  - If the element accessed is also a list, an additional indexing can be used.

```
>>> bilgi = [["Ali","Demir"],[[["Atatürk Cad.", "24"],
"06100"],"Ankara"]]
>>> print(bilgi[0])
['Ali','Demir']
>>> print(bilgi[0][1])
Demir
>>>  print(bilgi[1][0][1])
06100
```

# Tuples

- A tuple is an "immutable" data type.
- The tuples are defined similar to lists but "()" is used instead of "[]".
- The access to the elements is similar to that of lists.
- The advantages of tuple over lists:
  - Tuples are in general faster to process
  - Minimizes the programming bugs since they are immutable
  - Tuples as immutable types can be used as "keys" in "dictionary" data type.

# Tuples

- The elements of a tuple cannot be changed
- "Slicing" is similar to that of lists

```
>>> t = ("Lists", "and", "tuples")
>>> t[0]
'Lists'

>>> t[1:3]
('and', 'tuples')

>>> t[0]="new value"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

# Concatenation and Repetition in Sequential Data Types

- Sequential data types in Python can be concatenated with "+" operator like we in strings:

```
>>> a = [1,2,5,4]
>>> b = [8,14,9]
>>> c = [45,10,6]
>>> a + b + c
[1, 2, 5, 4, 8, 14, 9, 45, 10, 6]
```

- Similarly, repetition is done through "*" operator

```
>>> a*4
[1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4]
```

# Checking the presence of a specific element in sequential data types

- To check whether an element is contained in a sequential data type, "in" keyword/operator can be used:

```
>>> a = [1,2,5,4]
>>> 2 in a
True
>>> 7 not in a
True
>>> b = ("Ankara","İzmir","İstanbul")
>>> 'Ankara' in b
True
>>> 'Adana' not in b
True
```

# Shallow/Deep Copy Operations in Sequential Data Types

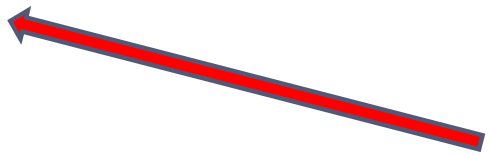
- When a new value is assigned to a variable, instead of modifying the data at the current memory address, a memory address is assigned to the variable and data is placed at the new memory address.

```
>>> x = 3
>>> y = x
>>> print(id(x), id(y))
1616756784 1616756784
>>> y = 4
>>> print(id(x), id(y))
1616756784 1616756800
>>> print(x,y)
3 4
```

# Shallow/Deep Copy Operations in Sequential Data Types

- Such phenomenon is also valid for sequential data types.

```
>>> colors  = ["red","blue","green"]
>>> colors2 = colors
>>> print(id(colors),id(colors2))
4317096 4317096
>>> colors2 = ["orange","brown"]
>>> print(colors,colors2)
['red', 'blue', 'green'] ['orange', 'brown']
>>> print(id(colors),id(colors2))
4317096 33918808
```



A new memory address is assigned!

# Shallow/Deep Copy Operations in Sequential Data Types

- Such phenomenon is also valid for sequential data types.

```
>>> colors  = ["red","blue","green"]
>>> colors2 = colors
>>> print(id(colors),id(colors2))
4317096 4317096
>>> colors2[0] = "orange"
>>> print(colors,colors2)
['orange', 'blue', 'green'] ['orange', 'blue','green']
>>> print(id(colors),id(colors2))
4317096 4317096
```

The elements of variable "colors" is also changed!

They have the same memory address!

# Shallow/Deep Copy Operations in Sequential Data Types

- To overcome such problems a special "copying" operation is needed. One such method is the "**shallow copy**"

```
>>> colors = ["red","blue","green"]
>>> colors2 = colors[:]
>>> print(id(colors),id(colors2))
2678696 10456760
>>> colors2[0] = "orange"
>>> print(colors,colors2)
['red', 'blue', 'green'] ['orange', 'blue', 'green']
```

Slicing operator! (shallow copy)

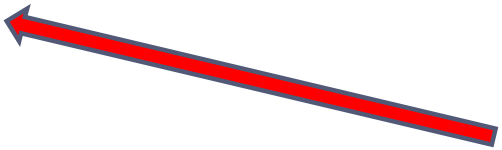
Instead of sharing a common memory address, a new memory address is assigned to the second variable!

# Shallow/Deep Copy Operations in Sequential Data Types

- If the sequential type already contains another sequential type, even the shallow copy is not sufficient:

```
>>> colors = [["red","blue"],"green"]
>>> colors2 = colors[:]
>>> print(id(colors),id(colors2))
10473840 2678696
>>> colors2[0][0] = "orange"
>>> print(colors,colors2)
[['orange', 'blue'], 'green'] [['orange', 'blue'], 'green']
```

Different addresses are assigned!



The element of the variable "colors" is also changed!

# Shallow/Deep Copy Operations in Sequential Data Types

- If the sequential type already contains another sequential type, a "**deep copy**" operation is needed.
- To perform a deep copy, deepcopy function from "deepcopy" module is imported.

```
>>> from copy import deepcopy
>>> colors = [["red","blue"],"green"]
>>> colors2 = deepcopy(colors)
>>> colors2[0][0] = "orange"
>>> print(colors,colors2)
[['red', 'blue'], 'green'] [['orange', 'blue'], 'green']
```

# References


1 Wentworth, P., Elkner, J., Downey, A.B., Meyers, C. (2014). How to Think Like a Computer Scientist: Learning with Python (3nd edition).
2 Pilgrim, M. (2014). Dive into Python 3 by. Free online version: DiveIntoPython3.org ISBN: 978-1430224150.
3 Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.
4 Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.
5 Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/.
6 Millman, K.J., Aivazis, M. (2011). Python for Scientists and Engineers, Computing in Science & Engineering, 13, 9-12.
7 John D. Hunter (2007). Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95.
8 Travis E. Oliphant (2007). Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20.
9 Goodrich, M.T., Tamassia, R., Goldwasser, M.H. (2013). Data Structures and Algorithms in Python, Wiley.
10 http://www.diveintopython.net/
11 https://docs.python.org/3/tutorial/
12 http://www.python-course.eu
13 https://developers.google.com/edu/python/
14 http://learnpythonthehardway.org/book/