CORVINUS UNIVERSITY of BUDAPEST

Python seminar 2.

Work with simple data types

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Creating int, float, bool and str type variables
- Constants in python
- Implicit and explicit type conversion
- Printing the variable values
- Format() method, f-strings, r-strings
- Asking for variable values
- Arithmetic operators and expressions

CORVINUS UNIVERSITY of BUDAPEST

Variables

A variable is a symbol associated with a memory address that contains a value. The value can be of any data type and can be changed.

- New variables can be created by assignment statements: variable_name = value
- Variable name have to begin with an underscore or a letter, can contain letters and digits.
- Variable name can not contain spaces
- Variable name is case-sensitive
- Variable name must be different from Python keywords
- The type of a variable is determined by the associated value
- If the value of a variable changes, its type may change as well

CORVINUS UNIVERSITY of BUDAPEST

Creating simple type variables

In [ ]:
```
#int type variable

x = 25
```

In [ ]:
```
#float type variable

y = 5.5
```

In [ ]:
```
#string type variables

s = "Hello"
t = 'World'
```

In [ ]:
```
#bool type variables

b = True
c = False
```

In [ ]:
```
#variable reassignment

a = 10
a = 'apple'
a = 4.4
```

CORVINUS UNIVERSITY of BUDAPEST

Constants

Constants are variable that cannot be changed

- In Python there is no dedicated syntax for defining constants
- But, there is a naming convention for constants
  - Use only capital letters
  - Use underscore to separate words
  - i.e. DEFAULT_TIMEOUT = 10
- Constants are often declared in different modules

CORVINUS UNIVERSITY of BUDAPEST

Type() function

The type() function is used to get the type of a variable or constant

In [20]:
```
#int type variable
x = 25
print(type(x))
#float type variable
y = 5.5
print(type(y))
#string type variables
s = "Hello"
print(type(s))
#bool type variables
b = True
print(type(b))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

In [21]:
```
#variable reassignment
a = 10
print(type(a))
a = 'apple'
print(type(a))
a = 4.4
print(type(a))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
```

CORVINUS UNIVERSITY of BUDAPEST

Explicit type conversion

Explicit type conversion (typecasting) takes place when the programmer defines the data type using in-built functions.

In [47]:
```
a = 5
print(type(a))
a = float(a)          # float() function converts to float if possible
print(type(s))
```

```
<class 'int'>
<class 'str'>
```

In [48]:
```
b = 23
print(type(b))      # int() function converts to int if possible
b = int(b)
print(type(b))
```

```
<class 'int'>
<class 'int'>
```

In [49]:
```
c = 10
print(type(c))
c = str(c)          # str() function converts to str
print(type(c))
```

```
<class 'int'>
<class 'str'>
```

Implicit type conversion

Implicit type conversion takes place when the interpreter modifies the data type

In [39]:
```
a = 25
print(type(a))
c = 30.0
print(type(c))
x = c / a
print(type(x))
```

```
<class 'int'>
<class 'float'>
<class 'float'>
```

CORVINUS UNIVERSITY of BUDAPEST

Print the variable values

Print() function can also display values of variables

Examples

```
In [2]: x = 15
        y = 20
        print(x)
        print(y)
```

```
15
20
```

```
In [1]: name = "Joe"
        age = 50
        is_student = True
        print("Name: ", name, "Age: ", age, "Is_student: ", is_student)
```

```
Name:  Joe Age:  50 Is_student:  True
```

CORVINUS UNIVERSITY of BUDAPEST

Format() method

Format() method returns a formatted string with the value passed as a parameter in the placeholder position.

Examples

```
In [4]: language = "Python"
        print("{} is our programming language".format(language))
```

```
Python is our programming language
```

```
In [3]: name = "Joe"
        age = 50
        is_student = True
        print("Name: {0}, Age: {1}, Is_student: {2}".format(name, age, is_student))
```

```
Name: Joe, Age: 50, Is_student: True
```

```
In [8]: print("5 * 5.25 is {result: .3f}".format(result = 5*5.25))
```

```
5 * 5.25 is  26.250
```

More examples

f-strings

f-strings contain a leading f character preceding the string and can be formatted in a similar way as if we where using the Format() method

Examples

In [2]:
```
language = "Python"
print(f"{language} is our programming language")
```

```
Python is our programming language
```

In [4]:
```
name = "Joe"
age = 50
is_student = True
print(f"Name: {name}, Age: {age}, Is_student: {is_student}")
```

```
Name: Joe, Age: 50, Is_student: True
```

In [12]:
```
print(f"5 * 5.25 is {5 * 5.25: .3f}")
#or
result = 5 * 5.25
print(f"5 * 5.25 is {result: .3f}")
```

```
5 * 5.25 is  26.250
5 * 5.25 is  26.250
```

CORVINUS UNIVERSITY of BUDAPEST

r-strings

r-strings (raw strings) contain a leading r character preceding the string and allow us the use \ character without interpreting it as escape sequece.

Examples

In [1]:
```
path = r"c:\users\public"
print(path)
```

```
c:\users\public
```

In [4]:
```
#compare
print("Today\nis\nWednesday")
print(r"Today\nis\nWednesday")
```

```
Today
is
Wednesday
Today\nis\nWednesday
```

In [8]:
```
#convert string to r-string using repr() function
s = "Hello World\n"
print(s)
s2 = repr(s)
print(s2)
```

```
Hello World

'Hello World\n'
```

Asking variable value

Input() function allows user input

- Its parameter is a message before the input
- It returns a string value therefore often type conversion is required

Examples

```
In [2]: x = input("Name: ")
        print(x)
        print(type(x))
```

```
Name: Joe
Joe
<class 'str'>
```

```
In [4]: #compare
        y = input("Age: ")
        print(y)
        print(type(y))

        y = int(input("Age: "))
        print(type(y))
```

```
Age: 25
25
<class 'str'>
Age: 25
<class 'int'>
```

```
In [9]: z = input("Are you a student? (yes or no) \n")
        z = bool(z)
        print(z)
```

```
Are you a student? (yes or no)
yes
True
```

CORVINUS UNIVERSITY of BUDAPEST

Arithmetic operators

Operator	Description	Example
+	Adds two operands	x + y
-	Subtracts two operands	a – 10
*	Multiplies two operands	b * 25
/	Divides the first operand by the second	12 / 5 (result: 2.4)
//	Divides the first operand by the second, rounded to the next smallest whole number (floor division)	12 / 5 (result: 2)
%	Returns the remainder of the division	12 % 5 (result: 2)
**	First operand is raised to the power of second operand	2 ** 5 (result: 32)

Arithmetic expressions

An arithmetic expression is a combination of arithmetic operators, numeric values and optionally parenthesis.

- In case of multiple operators, operator precedence define the order of evaluation.
- If the precedence are the same, the evaluation takes place from left to right

Examples

```
In [1]: x = 3  + 2 * 5
        print(x)

        13

In [2]: y = 2 ** 4 + 3
        print(y)

        19

In [4]: z = (-5 + 16 ** 0.5)/(2 * 0.5)
        print(z)

        -1.0
```

Precedence	Operators
1.	(), []
2.	**
3.	+, - (sign operators)
4.	*, /, //, %
5.	+, -

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 3.

Conditional branch in Python

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Logical expressions and operators
- Comparison and assignment operators
- Precedence of operators
- Binary selection with „if - else” statement
- Unary selection
- Nested conditionals
- Chained conditionals
- Match-case statement
- Ternary operator

CORVINUS UNIVERSITY of BUDAPEST

Logical operators

Logical operators operate on logical (boolean) expressions to combine the results into a single boolean value.

- not: True if and only if the operand is False
- and: True if and only if both operands are True
- or: True if and only if at least one of the operands is True

Truth table

X	Y	X and Y	X or Y	not X	not Y
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

CORVINUS UNIVERSITY of BUDAPEST

Logical expressions

A logical expression is an expression that evaluates to a boolean value.

Examples

In [4]:
```
a = 4
b = 3
print(a > 3)
print(b < 5)
```

```
True
True
```

In [8]:
```
x = 10
y = 8
a = (x == 10)
b = (y != 8)
print(a and b)
print(a or b)
print(not a)
```

```
False
True
False
```

In [11]:
```
a = True
b = False
c = False
r = (a and b) and (b or c)
print(r)
```

```
False
```

CORVINUS UNIVERSITY of BUDAPEST

Comparison operators

Comparison operators compare the value of two variables or expressions

Operator	Description	Example
<	Less than	x < 5
>	Greater than	y > 3
<=	Less than or equal to	x + 5 <= y
>=	Greater than or equal to	y >= x - 2
==	Equal	x == 3
!=	Not equal	x != y

CORVINUS UNIVERSITY of BUDAPEST

Assignment operators

Assigment operators are used to assign values to variables

Operator	Example	Meaning
=	x = 10	
+=	x += 10	x = x + 10
-=	y -= 5	y = y – 5
*=	z *= 2	z = z * 2
/=	a /= 3	a = a / 3
%=	b %= 5	b = b % 5
//=	c //= 5	c = c // 5

CORVINUS UNIVERSITY of BUDAPEST

Precedence of operators

Precedence	Operators
1.	(), []
2.	**
3.	+, - (sign operators)
4.	*, /, //, %
5.	+, -
6.	<, >, ==, !=, <=, >=
7.	not
8.	and
9.	or

Binary selection (if-else statement)

Binary selection means that depending on a condition (logical expression) we have two paths (branches) to continue the program.

```
if condition :
    command(s)*
else:
    other command(s)*
```

* The indent is very important

Examples

```
In [15]: x = int(input("give me a whole number other than zero"))
         if x > 0:
             print(f"{x} is positive")
         else:
             print(f" {x} is negative")

         give me a whole number other than zero55
         55 is positive
```

```
In [14]: a = 10
         b = int(input("enter a whole number"))
         if b == 0:
             print("the divison can not be performed")
         else:
             print(f"the quotiens of {a} and {b}: {a/b}")

         enter a whole number5
         the quotiens of 10 and 5: 2.0
```

Unary selection (if statement)

In case of unary selection there is only one branch of execution, which is triggered when the condition is true.

```
if  condition :
    command(s)*
```

Examples

```
In [18]: x = int(input("give me a whole number other than zero "))
         if x % 5 == 0:
             print(f"{x} can be divided by 5")
```

```
give me a whole number other than zero 20
20 can be divided by 5
```

```
In [21]: first_name = input("first name: ")
         last_name = input("last name: ")
         x = input("Do you have a middle name? (yes/no)")
         if x == "yes":
             middle_name = input("middle name: ")
             print(f"hello {first_name} {middle_name} {last_name}")
```

```
first name: Joe
last name: White
Do you have a middle name? (yes/no)yes
middle name: Peter
hello Joe Peter White
```

* The indent is very important

Nested conditionals

A selection can be nested within another

```
if condition :
        command(s)*
else:
        if condition2:
                commands2*
        else:
                other commands
```

Example

```
In [22]: x = int(input("enter a whole number"))
         if x > 0:
             print(f"{x} is positive")
         else:
             if x == 0:
                 print(f"{x} is zero")
             else:
                 print(f"{x} is negative")
```

```
enter a whole number-14
-14 is negative
```

* The indent is very important

Chained conditionals

The nested selection can also be implemented in other way combining the else and if statments into elif**.

```
if condition :
        command(s)*
elif condition2:
        commands2*
else:
        other commands
```

* The indent is very important

** elif can be more than one

Example

```
In [23]: x = int(input("enter a whole number"))
         if x > 0:
             print(f"{x} is positive")
         elif x == 0:
                 print(f"{x} is zero")
         else:
                 print(f"{x} is negative")
```

```
enter a whole number-15
-15 is negative
```

Match-case statement

From Python 3.10 the match-case statement can implement the multidirectional selection.

```
match parameter_value:
        case value1:
                commands1
        case value2:
                commands2
…
        case valuen:
                commandsn
        case _*:
                other commands
```

*case_ means the default case

Example

```
In [25]: x = int(input("enter your grade"))
         match x:
             case 1:
                 print("Failed")
             case 2:
                 print("Sufficient")
             case 3:
                 print("Average")
             case 4:
                 print("Good")
             case 5:
                 print("Excellent")
             case _:
                 print("you entered a wrong number")
```

```
enter your grade4
Good
```

Ternary operator

Ternary operator (conditional expression) is a concise way to write simple binary selection in a single line of code.

result_if_true **if** condition **else** result_if_false

Example

```
In [26]: x = 5
         print("Páros" if x % 2 == 0 else "Páratlan")

         Páratlan
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 4.

Loops in Python

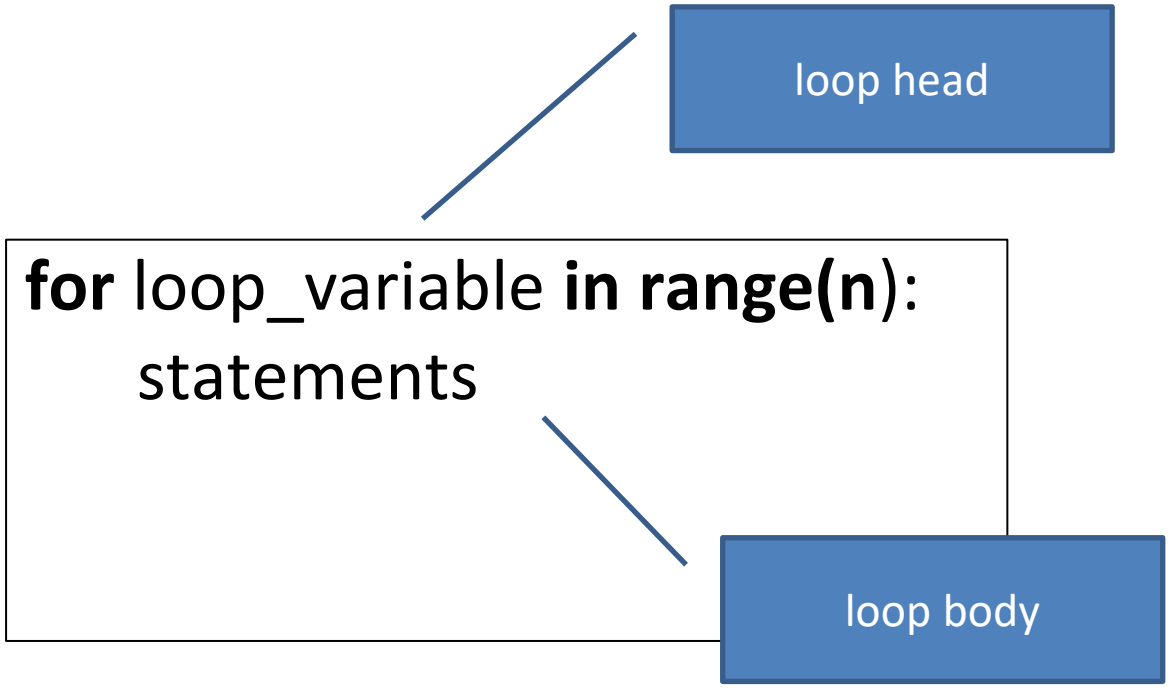
CORVINUS UNIVERSITY of BUDAPEST

Agenda

- For loop over a range
- While loop
- Emulate post-test loop
- Break, continue, return, and pass statements
- Random numbers
- Str type variables as a sequence of characters
- String operators

Loop over a range (for loop)

The for loop tells the program to execute specific program lines repeatedly.
The range() function specifies how many times the code inside the for loop will be executed.



The indent is very important

Examples

```
In [2]: for i in range(5):
            print("Hello World")

Hello World
Hello World
Hello World
Hello World
Hello World
```

```
In [1]: for i in range(3):
            print(i)

0
1
2
```

The range() function

The range() function also has optional parameters. With them the function returns a sequence of numbers within a given range.

range(start, end, step)

- **start:** start value of the range
- **end:** next value after the end value of the range
- **step:** the difference of the sequence

The indent is very important

Examples

```
In [3]: for i in range(3, 6):
            print(i)
```

```
3
4
5
```

```
In [4]: for i in range(1, 10, 2):
            print(i)
```

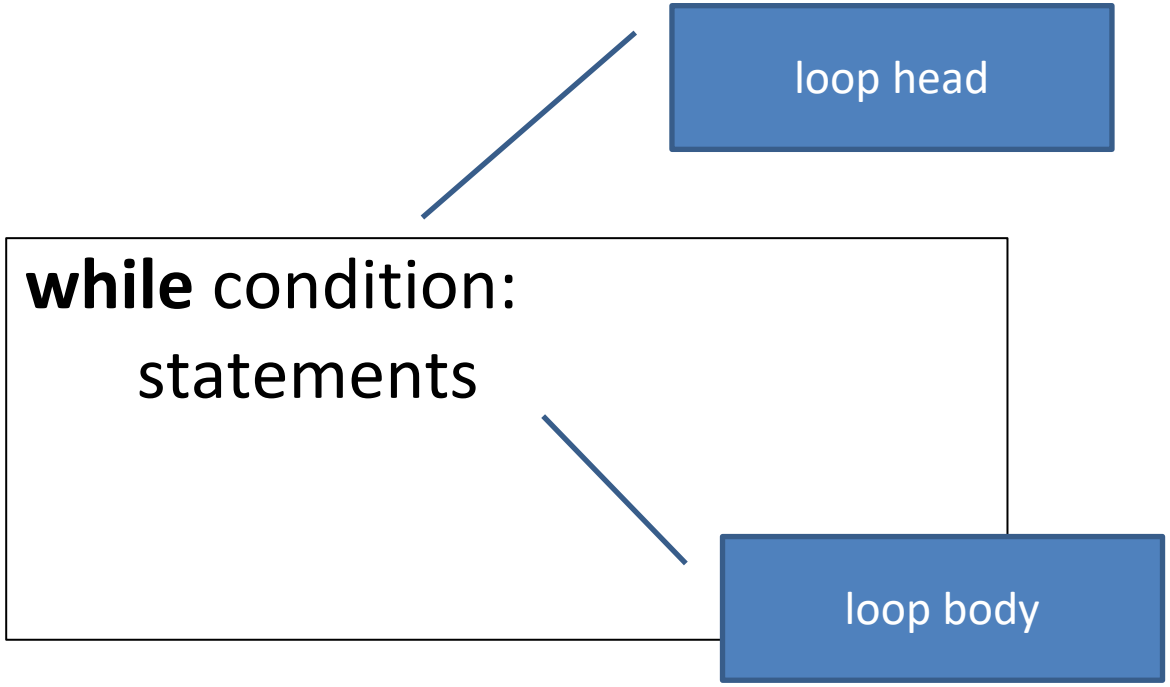
```
1
3
5
7
9
```

```
In [6]: # we can use _ as a loop variable if its value is not relevant
        for _ in range(5):
            print("Hello")
```

```
Hello
Hello
Hello
Hello
Hello
```

Pre-test loop (While loop)

The while loop executes a set of statments as long as a condition is true.



The indent is very important

Examples

```
In [7]: i = 1
        while i < 5:
            print(i)
            i = i + 1
```

```
1
2
3
4
```

```
In [8]: n = int(input("Enter a positive whole number "))
        i = 0
        while i <= n:
            i = i + 1
            if n % i == 0:
                print(i)
```

```
Enter a positive whole number 10
1
2
5
10
```

Nested loops

Within a loop (outer loop) we can place another loop (inner loop). It also works with for loops or while loops, moreover, we can also mix them.

outer loop

```
for loop_variable1 in range(n):
    statements1
    for   loop_variable2 in range(m)f:
        statements2
```

Inner loop

The indent is very important

Example

```
for i in range(1, 10):
    print(f"divisors of {i}:", end = " ")
    for j in range(1, i + 1):
        if i % j == 0:
            print(j, end = " ")
    print("")
```

```
divisors of 1: 1
divisors of 2: 1 2
divisors of 3: 1 3
divisors of 4: 1 2 4
divisors of 5: 1 5
divisors of 6: 1 2 3 6
divisors of 7: 1 7
divisors of 8: 1 2 4 8
divisors of 9: 1 3 9
```

Break* and continue

- break statement allows us to break out of the loop regardless of the loop condition.
- With the continue statement we can immediately jump to the next iteration by skipping the rest of the iteration.

```
for loop_variable in range(1, 10):
    statements
    continue
    skipped statements
```

Go to next iteration

```
while condition:
    statements
    break
    skipped statements
```

Exit from the loop

It is recommended to avoid using break or continue!

* Similar statement: return (we will learn it in a few weeks)

Example

```
#write a pogram, that display all even numbers starting from 2
#stop the program if the current number is divisible by 10

i = 1
more = True
while more:
    i = i + 1
    if i % 2 == 1:
        continue
    else:
        print(i)
    if i % 10 == 0:
        break
```

```
2
4
6
8
10
```

The pass statement

The pass statement is a null statement. It is useful when we will write later the implementation of a program block such as a branch or a loop.

```
if condition:
    pass        #or statements
else:
    pass        #or statements
```

```
while condition:
    statements
    pass
```

The indent is very important

Examples

```
i = int(input("enter a positive whole number"))
if i > 0:
    print("Thank you")
else:
    pass
```

```
enter a positive whole number-10
```

```
n = 10
while n < 100:
    n = n + 1
    pass
```

Post-test loop emulation

Post-test loop does not exist in Python, but we can emulate it.

```
while always_true_condition:
        statements
        if condition:
            break
```

```
logic_variable = true
while logic_variable:
        statements
        if condition:
            logic_variable = false
```

The indent is very important

Examples

```
# this works, but not recommended
while True:
    n = int(input("Enter a whole number between 1 and 100 "))
    if n >= 1 and n <= 100:
        print("Thank you")
        break
```

```
Enter a whole number between 1 and 100 877
Enter a whole number between 1 and 100 0
Enter a whole number between 1 and 100 2
Thank you
```

```
# this solution is better
more = True
while more:
    n = int(input("Enter a whole number between 1 and 100 "))
    if n >= 1 and n <= 100:
        more = False
        print("Thank you")
```

```
Enter a whole number between 1 and 100 877
Enter a whole number between 1 and 100 0
Enter a whole number between 1 and 100 2
Thank you
```

Random numbers

We can generate random numbers by using the random module and one function out of the followings:

- random.random() returns a float number between 0 and 1
- random.randint(a, b) returns a random integer value between a and b
- random.randrange(start, end, step) returns a random integer between start and end (end is not included) with step

Examples

```
import random
x = random.random()
print(f"{x :.4f}")
```

```
0.8384
```

```
import random
x = random.randint(1, 5)
print(x)
```

```
1
```

```
import random
x = random.randrange(2, 8, 3)
print(x)
```

```
5
```

Str type variables as a sequence of characters

Index 2	-10.	-9.	-8.	-7.	-6.	-5.	-4.	-3.	-2.	-1.
Index	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
Char	J	O	H	N		W	H	I	T	E

An str type variable can also be interpereted as a sequence of characters.

- Every character is directly accessible with the index operator (see next slide)
- We can access a substring of the str type variable with the slice operator (see next slide)
- Character numbering starts from zero

Examples

```
name = "John White"
first_char = name[0]
last_char = name[9]  # or name[-1]
print(f"first char is {first_char}, last char is {last_char}")
```

```
first char is J, last char is e
```

```
name = "John White"
left_six_chars = name[0:6]
print(f"the first six chars are {left_six_chars}")
```

```
the first six chars are John W
```

```
name = "John White"
last_two_chars = name[-2: ]
print(f"last two chars are {last_two_chars}")
```

```
last two chars are te
```

Warning, all variables of str type are immutable!

s = ”Hello”
s[1] = ”a” #error!!!

CORVINUS UNIVERSITY of BUDAPEST

String operators

Operator	Description	Example	Result
+	Concatenation	"John" + " " + "White"	"John White"
*	Repetition operator	"John White " *2	"John White John White "
[]	Indexing operator	"John White"[2]	"h"
[a : b]*	Slicing operator	"John White"[2: 5]	"hn "
>	Greater than	"John" > "White"	False
<	Less than	"apple" <"pear"	True
==	Equal	s1 = "ite" s2 = "White" s1 == s2	False
!=	Not equal	s1 = "ite" s2 = "White" s1 != s2	True
in	Membership	s1 = "i" s2 = "White" s1 in s2	True

* With optional parameter: [a: b: c], where c is the value of step

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 5.

Compound data types in Python I.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Sequences
  - String
  - List
  - Tuple
- Dictionary
- Set

CORVINUS UNIVERSITY of BUDAPEST

Sequence

Sequence is a collection of items or elements that are ordered and can be iterated over.

The most common used sequence types:

- String e.g. ”Hello World”
- List e.g. [1, 3, 4, 5, ’Hello’]
- Tuple e.g. (’one’, ’two’, ’three’, 4)

Sequences belong to iterable object. It is a broader category that includes all objects that can be looped over an iterator. It also includes sets and dictionaries.

CORVINUS UNIVERSITY of BUDAPEST

Common features of sequences

- **Indexing:** items can be accessed by their position (index) whitch starts at 0 (first element) and goes up to number of items – 1
- **Slicing:** sequences supports slicing operator to get sub-sequences
- **Iteration:** we can access each items in the sequence one by one
- **Concatenation:** sequences can be combined into a new sequence with the + operator
- **Repetition:** sequences can be repeated using the * operator
- **Membership testing:** we can check if an item exists in a sequence using the in operator
- **Length:** we can determine the number of items using the len() function
- **Ordered:** items are stored and accessed in the same order they where added

CORVINUS UNIVERSITY of BUDAPEST

Common sequence methods and functions

Method/Function	Description	Example	Result
len()	Returns the number of items in the sequence	t = (1, 4, 7) --> len(t)	3
min()*	Returns the minimum value in a sequence	l = [4, 1, -3] --> min(l)	-3
max()*	Returns the maximum value in a sequence	l = [4, 1, -3] --> max(l)	4
sum()**	Returns the sum of items in a sequence	l = [4, 1, -3] --> sum(l)	2
count()	Returns the number of occurences in a specific item in a sequence	s = 'hello world' --> count('o')	2
sorted()	Returns a new sorted list	l = [1, 4, -2] --> sorted(l)	[-2, 4, 1]
reversed()	Returns a reverse iterator for a sequence	for c in reversed('hello'): print(c)	Olleh
enumerate()	Returns items and values as an iterator	for i, v in enumerate([3, 4]): print(i, v)	0 3 1 4

* If we have comparable items

** If we have numbers in the sequence

String as a sequence (see also seminar 4)

A string variable (str) can also be considered as a sequence of chars

Iteration over a string:

- **for** i **in** range(0, **len**(string)):
  do something with i-th char

- **for** ch **in** string:
  do something with the current char

```
In [3]: #iterate over index
        s = "Hello World"
        for i in range(0, len(s)):
            print(s[i], end = '')

Hello World
```

```
In [2]: #iterate over the string
        s = "Hello World"
        for c in s:
            print(c, end = '')

Hello World
```

CORVINUS UNIVERSITY of BUDAPEST

Special string methods*

Method	Description	Example: s = ”Hello World”	Result
s.lower()	Converts all the characters in a string to lowercase	s.lower()	hello world
s.upper()	Converts all the characters in a string to uppercase	s.upper()	HELLO WORLD
s.find()	Returns the index of a specified substring in the string	s.find(”lo”, s)	3
s.replace()	Replaces an old sutbstring with a new one	s.replace(”World”, ”Everyone”)	Hello Everyone
s.strip()	Removes whitespaces (or a specified substring) at the beginning and at the end of the string	” Hello World ”.strip()	Hello World
s.join()	Joins items of a sequence separated by a separator	”-”.join(s)	H-e-l-l-o- -W-o-r-l-d
s.isdigit()	Returns True, if all characters in string are digits	s.isdigit()	False
s.isalpha()	Returns True, if all characters in string are alphabetic		True

*only the more common ones

Attention, these methods always return a new string, the original one does not change!

List

A **list** is a sequential collection of Python data values, where each value is identified by an index.

Lists are similar to strings, but

- List items can be of different types
- List items can have any types
- List items are enclosed in square brackets
- Lists are mutable

Examples:

```
l = [1, 2, 3, 4]

l2 = ['1', '2', 3, 4.4]

l3 = ['hello', 'world', 2023, ['one', 'two']]

print(l)

print(l2[2:4])

print(l3[1])
```

```
[1, 2, 3, 4]
[3, 4.4]
world
```

List type can be used as an array if the items are of the same type

Two dimensional list

A list of lists, where each element in the outer list is itself a list

Often used to represent

- Tables
- Grids
- Matrices

We need two indices to refer to the list elements:

- First index refers to the row
- Second index refers to the column

Examples:

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

print(matrix[0][1])
print(matrix[0])
print(matrix[0:1])
print(matrix[1: ])
matrix[1][1] = 10
print(matrix)
```

```
2
[1, 2, 3]
[[1, 2, 3]]
[[4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 10, 6], [7, 8, 9]]
```

CORVINUS UNIVERSITY of BUDAPEST

List comprehension

It is a concise and powerful way to create lists in Python

Basic syntax: newlist = [expression **for** item **in** iterable **if** condition]

```
squares = [x ** 2 for x in range(10)]  # list of squares of numbers between 0 and 9

odds = [x for x in range(10) if x % 2 == 0] # list of odd numbers between 0 and 9

words = ["apple", "banana", "cherry"]
upperwords = [x.upper() for x in words]

print(squares)
print(odds)
print(upperwords)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8]
['APPLE', 'BANANA', 'CHERRY']
```

CORVINUS UNIVERSITY of BUDAPEST

Common list methods

Method	Description	Example: l = [1, 2, 3]	Result
append(item)	Appends a new item to the end of the list	l.append(4)	[1, 2, 3, 4]
insert(index, item)	Inserts a new item at the given index	l.insert(1, 4)	[1, 4, 2, 3]
remove(item)	Removes the first occurence of the item from the list	l.remove(2)	[1, 3]
pop(index)	Removes and returns the item at the specified index. If no index is given, then removes and returns the last item.	l.pop(0)	[2, 3]
index(item, start, end)	Returns the index of the first occurence of the given item within start and end range	l.Index(3)	2
count(item)	Returns the number of times the given item appears in the list	l.count(2)	1
clear()	Removes all items from the list	l.clear()	[ ]
copy()	Makes a copy of the list	k = l.copy()	[1, 2, 3]
reverse()	Modifies the list to be in reverse order	l.reverse()	[3, 2, 1]

Tuple

Tuple is a list-like data structure but it is immutable

- The elements of a tuple are enclosed in parantheses
- Tuple can be faster than list in certain operations
- We use tuple when the collection of items should remain constants such as coordinates, settings
- Tuple has only those list methods that do not modify values of tuple (i.e. not exist: append(), pop(), clear() etc)

Examples:

```
my_tuple = (4, 22, 11)
my_tuple2 = ("hello", "world", 2023)

print(my_tuple[1])
print(my_tuple2[1:2])
```

```
22
('world',)
```

CORVINUS UNIVERSITY of BUDAPEST

Set

A set is an unordered collection of unique elements

Features:

- Items are within curly braces
- No specific order
- No duplicates
- No indices
- Sets are mutable

For methods see the next slide

```
s = {3, 2, 4}
s2 = {3, 5}
s3 = {1, 4, 7}

s.add(8)
s2.add(3)   # not duplicates!
s3.pop()

print(s, s2, s3)
print(s.union(s2))
print(s.intersection(s3))
print(s2.difference(s3))
```

```
{8, 2, 3, 4} {3, 5} {4, 7}
{2, 3, 4, 5, 8}
{4}
{3, 5}
```

CORVINUS UNIVERSITY of BUDAPEST

Common set methods

Method	Description	Example: s = {3, 2, 4}	Result
add(item)	Adds an item to the set	s.add(5)	{3, 2, 4, 5}
discard(item)	Removes the given items from the set	s.discard(3)	{2, 4}
pop()	Removes an item from the set	s.pop()	{2, 3} or {2, 4} or {3, 4}
clear()	Removes all items from the set	s.clear()	{ }
difference(set2)	Returns a set containing the difference between two or more sets	s.difference({3, 5})	{2, 4}
union(set2)	Returns a set containing the union of two or more sets	s.union({3, 5})	{3, 2, 4, 5}
intersection(set2)	Returns a set containing the common items of two or more sets	s.intersection({3, 5})	{3}

Dictionary

Dictionaries are collections of key-value pairs

Features:

- Unordered
- Mutable (but keys are not!)
- No duplicate keys
- Efficient key lookup
- Used for mapping one value (key) to another value (value)

For methods see the next slide

```
person = {"name": "John", "age": 30, "city": "New York"}

print(person.values())
print(person.keys())
print(person.get("name"))
print(person["name"])

person["job"] = "Teacher"  # add a new key-value pair
person["age"] = 35  # modify a value

print(person)
```

```
dict_values(['John', 30, 'New York'])
dict_keys(['name', 'age', 'city'])
John
John
{'name': 'John', 'age': 35, 'city': 'New York', 'job': 'Teacher'}
```

Dictionaries can be used to implement a record-like data structure too

CORVINUS UNIVERSITY of BUDAPEST

Common dictionary methods

Method	Description	Example: d = {"name": "John", "age": 30}	Result
copy()	Returns a copy of the dictionary	d.copy()	{"name": "John", "age": 30}
pop(key)	Removes the item with the given key	d.pop("age")	{"name": "John”}
clear()	Removes all items from the dictionary	d.clear()	{ }
get(key)	Returns the value of the given key	d.get("age")	30
values()	Returns a list of all the values	d.values()	["John", 30]
keys()	Returns a list of all keys	d.keys()	["name", "age"]

Dictionary comprehension

It is a concise and powerful way to create dictionary in Python

Basic syntax: newdict = { key:value **for** (key, value) **in** iterable **if** condition}

```
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]

# zip() function pairs keys and values
myDict = { k:v for (k,v) in zip(keys, values)}
squares = {x: x**2 for x in range(5)}
odd_squares = {x: x**2 for x in range(5) if x % 2 == 1}

print(myDict)
print(squares)
print(odd_squares)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
dict_values([1, 2, 3, 4, 5]) dict_keys(['a', 'b', 'c', 'd', 'e'])
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 3: 9}
```

CORVINUS UNIVERSITY of BUDAPEST

Conversion between list, tuple, set, dictonary

Conversion can be done with the following constructor functions:
list(), tuple(), set(), dict()

```
mylist = [1, 2, 3, 4, 5]
mytuple = (3, 4, 5)
myset = {1, 4, 9}
mydict = {'name': "James", 'country': 'UK'}

# convert list, set and dict to tuple
tuple1 = tuple(mylist)
tuple2 = tuple(myset)
tuple3 = tuple(mydict.keys())
tuple4 = tuple(mydict.values())
print(tuple1, tuple2, tuple3, tuple4)

# convert tuple, set and dict to list
list1 = list(mytuple)
list2 = list(myset)
list3 = list(mydict.keys())
list4 = list(mydict.values())
print(list1, list2, list3, list4)
```

```
(1, 2, 3, 4, 5) (1, 4, 9) ('name', 'country') ('James', 'UK')
[3, 4, 5] [1, 4, 9] ['name', 'country'] ['James', 'UK']
```

```
# convert list, tuple, dict to set
set1 = set(mylist)
set2 = set(mytuple)
set3 = set(mydict.keys())
set4 = set(mydict.values())
print(set1, set2, set3, set4)

#convert list, tuple, set to dict
mylist2 = [('a', 1), ('b', 2)]
myset2 = {('a', 1), ('b', 2)}
mytuple2 = (('a', 1), ('b', 2))
dict1 = dict(mylist2)
dict2 = dict(myset2)
dict3 = dict(mytuple2)
print(dict1, dict2, dict3)
```

```
{1, 2, 3, 4, 5} {3, 4, 5} {'name', 'country'} {'James', 'UK'}
{'a': 1, 'b': 2} {'b': 2, 'a': 1} {'a': 1, 'b': 2}
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 6.

Basic algorithms I.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Decision
- Summarization
- Counting
- Selection of an item with a given property
- Linear search
- Maximum selection

Decision

Is there an element with property P in the sequence?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while i <= N and ¬P(x[i])
    i = i + 1
end loop
exists = (i <= N)
```

Example

```
: # Is there 0 among the elements?
  # P: x[i] == 0 --> not P: x[i] != 0
  x = [3, 1, 2, 8, 4]
  N = len(x)
  i = 0
  while i <= N - 1 and x[i] != 0:
      i = i + 1
  exists = (i <= N - 1)
  print(exists)
```

```
False
```

Summarization

What is the sum of the elements of the sequence?

N: number of elements, X: sequence
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
sum = 0
loop for i in 1..N
    sum = sum + x[i]
end loop
```

Example

```
# What is the sum of the elements?

x = [3, 1, 2, 8, 4]
N = len(x)
sum = 0
for i in range(0, N):
    sum = sum + x[i]
print(sum)
```

```
18
```

Counting

How many elements with property P are there in the sequence?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
count = 0
loop for i in 1..N
    if P(x[i]):
        count = count + 1
    end if
end loop
```

Example

```
# How many positive numbers are there in the sequence?
# P: x[i] > 0
x = [3, -1, 0, 8, 4]
N = len(x)
count = 0
for i in range(0, N):
    if x[i] > 0:
        count = count + 1
print(count)
```

```
3
```

Selection of an item with P

Which is the very first element with property P in the sequence? (assume there is one)

N: number of elements, X: sequence, P: a given property

In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while ¬P(x[i])
    i = i + 1
end loop
number = i
# element = x[i]
```

Example

```
# Which element of the sequence is London?
# P: x[i] == 'London' --> not P: x[i] != 'London'
x = ['Budapest', 'New York', 'Paris', 'London', 'Los Angeles']
i = 0
while x[i] != 'London':
    i = i + 1
number = i
print(number)
```

```
3
```

Linear search

Is there an element with property P in the sequence?
If there is one, which one is the first?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while i <= N and ¬ P(x[i])
   i = i + 1
end loop
exists = (i <= N)
If exists
    number = i
End if
```

Example

```python
# Is there 5 among the elements? If yes, which one?
# P: x[i] == 5 --> not P: x[i] != 5
x = [8, 4, 7, 9, 5, 11]
N = len(x)
i = 0
while i <= N-1 and x[i] != 5:
    i = i + 1
exists = (i <= N-1)
if exists:
    number = i
    print(number)
else:
    print("there is no 5 among the elements")
```

Maximum selection

Which is the biggest item in the sequence?

N: number of elements, X: sequence
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
max = 1
loop for i in 2..N
   if x[i] > x[max]
      max = i
end loop
```

Example

```
# Which is the biggest item in the sequence?
x = [8, 4, 7, 9, 5, 11]
N = len(x)
max = 0
for i in range(1, N):
    if x[i] > x[max]:
        max = i
print(max)
print(x[max])
```

```
5
11
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 6.

Basic algorithms I.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Decision
- Summarization
- Counting
- Selection of an item with a given property
- Linear search
- Maximum selection

Decision

Is there an element with property P in the sequence?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while i <= N and ¬P(x[i])
    i = i + 1
end loop
exists = (i <= N)
```

Example

```
: # Is there 0 among the elements?
  # P: x[i] == 0 --> not P: x[i] != 0
  x = [3, 1, 2, 8, 4]
  N = len(x)
  i = 0
  while i <= N - 1 and x[i] != 0:
      i = i + 1
  exists = (i <= N - 1)
  print(exists)
```

```
False
```

Summarization

What is the sum of the elements of the sequence?

N: number of elements, X: sequence
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
sum = 0
loop for i in 1..N
    sum = sum + x[i]
end loop
```

Example

```
# What is the sum of the elements?

x = [3, 1, 2, 8, 4]
N = len(x)
sum = 0
for i in range(0, N):
    sum = sum + x[i]
print(sum)
```

```
18
```

Counting

How many elements with property P are there in the sequence?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
count = 0
loop for i in 1..N
    if P(x[i]):
        count = count + 1
    end if
end loop
```

Example

```
# How many positive numbers are there in the sequence?
# P: x[i] > 0
x = [3, -1, 0, 8, 4]
N = len(x)
count = 0
for i in range(0, N):
    if x[i] > 0:
        count = count + 1
print(count)
```

```
3
```

Selection of an item with P

Which is the very first element with property P in the sequence? (assume there is one)

N: number of elements, X: sequence, P: a given property

In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while ¬P(x[i])
    i = i + 1
end loop
number = i
# element = x[i]
```

Example

```
# Which element of the sequence is London?
# P: x[i] == 'London' --> not P: x[i] != 'London'
x = ['Budapest', 'New York', 'Paris', 'London', 'Los Angeles']
i = 0
while x[i] != 'London':
    i = i + 1
number = i
print(number)
```

```
3
```

Linear search

Is there an element with property P in the sequence?
If there is one, which one is the first?

N: number of elements, X: sequence, P: a given property
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
i = 1
loop while i <= N and ¬ P(x[i])
   i = i + 1
end loop
exists = (i <= N)
If exists
    number = i
End if
```

Example

```python
# Is there 5 among the elements? If yes, which one?
# P: x[i] == 5 --> not P: x[i] != 5
x = [8, 4, 7, 9, 5, 11]
N = len(x)
i = 0
while i <= N-1 and x[i] != 5:
    i = i + 1
exists = (i <= N-1)
if exists:
    number = i
    print(number)
else:
    print("there is no 5 among the elements")
```

Maximum selection

Which is the biggest item in the sequence?

N: number of elements, X: sequence
In Python, element numbering starts at 0, not 1 → i in 0..N-1

Pseudocode:

```
max = 1
loop for i in 2..N
   if x[i] > x[max]
      max = i
end loop
```

Example

```
# Which is the biggest item in the sequence?
x = [8, 4, 7, 9, 5, 11]
N = len(x)
max = 0
for i in range(1, N):
    if x[i] > x[max]:
        max = i
print(max)
print(x[max])
```

```
5
11
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 7.

Basic algorithms II.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Copying
- Selection (elements with a given property)
- Separation
- Swap sort
- Minimum selection sort
- Bubble sort

Copying

Let's transform the input sequence with element-by-element processing!

N: number of elements, X: input sequence, y: output sequence, f: function of transformation
In Python, element numbering starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
loop for i in 1..N
    y[i] = f(x[i])
end loop
```

Example

```
# convert the values from cm to meters
# f(x) = x / 100
values_in_cm = [170, 155, 185, 168, 172]
n = len(values_in_cm)
values_in_m = [0]*n
for i in range(n):
    values_in_m[i] = values_in_cm[i] /100.0
print(values_in_m)
```

```
[1.7, 1.55, 1.85, 1.68, 1.72]
```

Selection (elements with P property)

Let's select the elements of the input sequece with property P (or their indexes) into a new sequence!

N: number of elements, X: input sequence, Y: output sequence, P: property
In Python, element numbering starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
j = 0
loop for i in 1..N
    if P(x[i])
        j = j + 1
        y[j] = x[i]      #or y[j] = i
    end if
end loop
```

Example

```
# Which numbers are divisible by 5 in the following series?
# P(z): z % 5 == 0
x = [170, 155, 185, 168, 172]
y = []
n = len(x)
j = -1
for i in range(n):
    if x[i] % 5 == 0:
        j +=  1
        y.append(0)
        y[j] = x[i]
print(y)
```

```
[170, 155, 185]
```

Separation

Let's separate the elements of a sequence into two new sequences depending on whether they have a property P or not.

N: number of elements, X: input sequence, y, z: output sequences, P: a given property

In Python, element indexing starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
j = 0
k = 0
loop for i in 1..N
   if P(x[i])
      j = j + 1
      y[j] = x[i]      #or y[j] = i
   else
      k = k + 1
      z[k] = x[i]      #or z[k] = i
   end if
end loop
```

Example

```
# Which numbers are divisible by 5, and which are not in the following series?
# P(z): z % 5 == 0
x = [170, 155, 185, 168, 172]
y = []
z = []
n = len(x)
j = -1
k = -1
for i in range(n):
    if x[i] % 5 == 0:
        j +=  1
        y.append(0)
        y[j] = x[i]
    else:
        k += 1
        z.append(0)
        z[k] = x[i]
print(y)
print(z)
```

```
[170, 155, 185]
[168, 172]
```

Swap sort

Go from left to right and compare each element with all the ones after it. If in one of the comparisons we find a smaller one than the given element, then let's swap the compared elements!

N: number of elements, X: sequence

In Python, element numbering starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
loop for i in 1..N-1
    loop for j in i+1..N
        if x[j] < x[i]
            swap(x[i], x[j])
        end if
    end loop
end loop
```

Example

```
x = [170, 155, 185, 168, 172]
n = len(x)
```

```
for i in range(n-1):
    for j in range(i+1, n):
        if x[i] > x[j]:
            x[i], x[j] = x[j], x[i]
print(x)
```

```
[155, 168, 170, 172, 185]
```

Minimum selection sort

Go from left to right and compare each element with all the ones after it. Let's swap the given element with the smallest one after it.

N: number of elements, X: sequence

In Python, element numbering starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
loop for i in 1..N-1
    min = i
    loop for j in i+1..N
        if x[j] < x[min]:
            min = j
    end loop
    swap(x[i], x[min])
end loop
```

Example

```
# Let's sort the following numbers in ascending order
# using minimum selection sort

x = [170, 155, 185, 168, 172]
for i in range(n - 1):
    min = i
    for j in range(i+1, n):
        if x[j] < x[min]:
            min = j
    x[i], x[min] = x[min], x[i]
print(x)
```

```
[155, 168, 170, 172, 185]
```

Bubble sort

Go from left to right and always compare adjacent elements. If the relationship between them is not appropriate, then we swap them. Let's repeat these steps so that we always skip the last element!

N: number of elements, X: sequence

In Python, element numbering starts at 0, not 1 ➔ i in 0..N-1

Pseudocode:

```
loop for i in N..2
    loop for j in 1..i-1
        if x[j] > x[j+1]:
            swap(x[j], x[j+1])
    end loop
end loop
```

Example

```
# Let's sort the following numbers in ascending order
# using bubble sort

x = [170, 155, 185, 168, 172]
for i in range(n - 1, 0, -1):
    for j in range(i):
        if x[j] > x[j+1]:
            x[j], x[j+1] = x[j+1], x[j]
print(x)
```

```
[155, 168, 170, 172, 185]
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 9.

Procedures and functions.
Modules.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- User-defined functions and procedures in Python
- Passing parameters
- Recursion
- Local and global variables
- Lambda functions
- Modules and packages

User-defined functions and procedures*

The (user-defined) function and the procedure are both a block of statements that perform a specific task. The function returns the result too, the procedure does not.

The syntax to declare a function or a procedure:

```
def  function_name(parameters):

     """ docstring """  #describes the function

     statements

     return expression
```

- At the procedures, the return statement is omitted
- The docstring is optional

Examples

```python
def is_whole_number(number): #this is a function
    # Check if the number is equal to its integer conversion
    return number == int(number)

print(is_whole_number(5.0))  # True
print(is_whole_number(5.5))  # False
```

```python
def duplicate_number(number):  #this is a procedure
    # Multiply the number by 2 to duplicate it
    number[0] = number[0] * 2
x = [5]
duplicate_number(x)
print(x) # [10]
```

*Functions and procedure play an important role in code reusability, just as methods, modules, packages, libraries, inheritance, decorators and compositions. (see some of them later)

Passing parameters

In Python, immutable data types (integers, floats, strings, tuples) are passed by value. In other cases, parameters are passed by reference.

- **Passing by value** means a copy of the original value. Any changes made to the parameter inside the function do not affect the original object outside the function.
- **Passing by reference** means that no copy of the original variable is created. So any changes made to the parameter inside the function will affect the original object outside the function.

Examples

```python
def modify_list(my_list): #passing parameter by reference
    my_list.append(42)

original_list = [1, 2, 3]
modify_list(original_list)
print(original_list)  # Output: [1, 2, 3, 42]
```

```python
def modify_string(s):  #passsing parameter by value
    s = "Hello, world!"

original_string = "Hi"
modify_string(original_string)
print(original_string)  # Output: "Hi"
```

Parameter types

The types of parameters:

- **Positional parameters**: the order of the parameters is important
- **Default parameter**: assumes a default value if the value is not provided in the function call
- **Keyword parameters**: the parameters are specified with names, the order is not important
- **Arbitrary parameters**: can pass a variable number of parameters

Examples

```python
def greet(name, greeting): #positional parameters
    return f"{greeting}, {name}!"

result = greet("Alice", "Hello")
print(result)  # Output: "Hello, Alice!"
```

```python
def greet(name, greeting="Hello"): #default parameters
    return f"{greeting}, {name}!"

result = greet("Bob")
print(result)  # Output: "Hello, Bob"
```

```python
def greet(name, greeting="Hello"): #keyword parameters
    return f"{greeting}, {name}!"

result = greet(name="Charlie", greeting="Hi")
print(result)  # Output: "Hi, Charlie"
```

```python
def sum_numbers(*args): #arbitrary parameters
    total = 0
    for num in args:
        total += num
    return total

result = sum_numbers(1, 2, 3, 4, 5)
print(result)  # Output: 15
```

CORVINUS UNIVERSITY of BUDAPEST

Recursion

Recursion refers to a technique where a function calls itself in order to solve a problem.

Instead of using loops to perform repetitive tasks …

- ❑ breaks down a problem into smaller, similar subproblems and solves them through successive calls to itself (recursion).
- ❑ This process continues until a base case is reached, at which point the function returns a result (termination).

Example

```python
def recursive_sum(n): # add numbers from 1 to n
    # Base case (termination)
    if n == 1:
        return 1
    # Recursion
    else:
        return n + recursive_sum(n - 1)

result = recursive_sum(5)
print(result)  # Output: 15 (1 + 2 + 3 + 4 + 5 = 15)
```

Local and global variables

- **Local variables** are defined within a specific function or block of code, and they are only accessible and visible within that function or block.
- **Global variables** are defined outside of any function or block and can be accessed from anywhere in the code, both inside and outside functions.

The **global** keyword indicates, that we are working with the global variable rather than creating a new local variable.

Examples

```
x = 10  # Global variable

def my_function():
    x = 20  # This creates a local variable 'x' within the function's scope
    print(x)

my_function()
# Output: 20 (Prints the local variable 'x' within the function's scope)

print(x)
# Output: 10 (Accesses the global variable 'x' outside the function)
```

```
def my_function2():
    global x
    x = 20  # This creates a local variable 'x' within the function's scope
    print(x)

my_function2()
print(x)    # x is avaible also outside of the function
```

```
20
20
```

Lambda functions

A lambda function is a small, anonymous, one-line function that can have any number of arguments (parameters) but can only have one expression.

The syntax to declare a lambda function:

lambda arguments: expression

Lambda function is often combined with the …

- map() function to apply the same operation to every item of an iterable
- filter() function to select items from an iterable
- reduce() function to perform cumulative operations on an iterable

```
add = lambda x, y: x + y
result = add(3, 5)
print(result)  # Output: 8
```

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]

numbers_below_4 = list(filter(lambda x: x if x < 4 else None, numbers))
print(numbers_below_4)
```

```
[1, 4, 9, 16, 25]
[1, 2, 3]
```

```
from functools import reduce

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

# Use reduce() with an initial value
product = reduce(lambda x, y: x * y, numbers, 1)
print(product)  # Output: 120 (1 * 2 * 3 * 4 * 5)
```

Lambda functions are used to solve only simple, short tasks.

Modules and packages

A module is a file containing Python code, including functions, classes, and variables, that can be used in other Python programs. Packages are directories containing one or more modules.

- **To create a module**, write the code in a .py file
- **To use the module**, you need to import it by the **import** modulname statement (without .py)
- You can **use an alias name** to make it easier the reference
- **You can access** the **modul elements** either by the **dot notation** or the **from** statement

Examples

```
import mymodule   #import a module

import mymodule as mm  #import with alias

result = mymodule.myfunction()  #dot notation

from mymodule import myfunction2() #from statement
result2 = myfunction2()
```

Module example

Module

```python
# my_module.py

# Function to calculate the square of a number
def square(x):
    return x * x

# Function to find the factorial of a number
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# A global variable
greeting = "Hello, World!"
```

Module usage

```python
# Import the entire module
import my_module

# Use functions and variables from the module
result1 = my_module.square(5)
result2 = my_module.factorial(4)
message = my_module.greeting

print("Square:", result1)
print("Factorial:", result2)
print("Message:", message)
```

The main module

When a Python script is executed, it is considered the "main" module.

- It is the starting point of the execution
- if __name__ == "__main__": allows you to specify code that should only run when the script is executed as the main module.
- It prevents code from running when the module is imported as a module in other scripts.

```
def hello_world():
    print("Hello, World!")

if __name__ == "__main__":
    hello_world()
    print("This code runs in the main module.")
```

In Jupyter Notebook, the concept of a "main module" is not as relevant or significant as in traditional Python scripts or modules.

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 10.

Compound data types II.
Text files.

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Files in Python
- Basic text file operations
- File pointer positioning
- Create, rename and delete a text file
- Useful string methods for handling text files
- Handling exceptions

Files in Python

A file is a structured collection of data that are stored on the disk.

The file's data structure can be various:

- **Sequential**. Data is stored in a linear manner from the beginning to the end. This is common in text files, where lines of text are stored one after another.
- **Hierarchical**. Data is organized in a tree-like fashion with parent-child relationships, such as JSON and XML
- **Tabular**. Dat is organized in a tabular structure, with rows and columns. Each line in the file corresponds to a record, and values are separated by commas (or other delimiters).
- **Binary**. Binary files contain raw binary data organized in a format specific to the application. For example, image files, audio files, and executable files.
- **Custom**. Data organizing way is defined by the application

CORVINUS UNIVERSITY of BUDAPEST

Text files

There are several concepts related to text file handling :

- **File Pointer.** The file pointer is a marker that represents the current position in the file. When you open a file, the file pointer is initially set to the beginning of the file. As you read or write data, the file pointer moves to keep track of the current position.
- **End-of-Line (EOL).** The end-of-line character represents the termination of a line in a text file. The specific character used as the end-of-line marker can vary between operating systems.
- **End-of-File (EOF).** The end-of-file marker indicates the end of the file. When reading a file, you can use the EOF marker to determine when you've reached the end of the file. In Python, this is typically detected by reaching an empty string when using read() or readline().
- **File Modes.** When opening a file, you specify a file mode that determines how the file should be opened (read, write, append, etc.). The default mode is the read.

File pointer positioning*

- **seek(offset,** whence) method changes the position of the file pointer**
  - **offset:** the number of bytes to move. A positive offset moves the pointer forward, a negative one moves it backward
  - whence: specifies the reference point.
    0 means the beginning of the file (default)
    1 means the current position
    2 means the end of the file
- **tell()** returns the current position of the file pointer

fruits.txt
apple orange
banana pear plum
melon

Examples

```
with open("fruits.txt", "rb") as f:
    f.seek(1)
    print(f.read(1)) #print the first char (from zero)
    f.seek(25) #print the 25th char (from zero, including eol chars)
    print(f.read(1))
    f.seek(-2, 2)
    print(f.read(1)) #print the penultimate char
    print(f.tell())  #the penultimate position
    f.seek(0, 2) #jump to the end of file
    print(f.tell())#the last position of the file
```

```
b'p'
b'r'
b'o'
37
38
```

* The seek method should be used with caution in write mode
** To move backward, the file must be opened in binary mode, i.e. using rb as opening mode.

CORVINUS UNIVERSITY of BUDAPEST

Basic text file operations and methods

Operation	Parameters	Description	Example
open()	**filename** mode* encoding	Open a text file with a given filename Open a text file with a given mode **r: only reading**, w: write, a: append, r+: reading and writing existing file, w+: reading and writing, a+: append and read Open a text file with a given encoding UTF-8 (most commonly used)	f = open("a.txt") f = open("a.txt", "r") f = open("a.txt", "r", "UTF-8")
close()	-	Closes the file	f.close()
read() readline() readlines()	n n n	Reads the whole file or reads n bytes from file into a string Reads one line or n bytes from file Reads each line or max. n bytes from file into a list	s = f.read() line = f.readline() content = f.readlines()
write() writelines()	text list	Writes the text into the file without adding any extra chars Writes the list into the file without adding any extra chars	f.write(mystring) f.writeline(mylist)
append()	text	Appends the text to the file	f.append(mytext)

* At w, w+ modes the existing content will be overridden, at r+ mode can be overridden!

Create, rename and delete a text file

- To create a new text file, you can use the **open()** function with mode 'w' (write). If the file already exists, it will be truncated; otherwise, a new file will be created.
- To rename a file, you can use the os.**rename()** function from the os module
- To delete (remove) a file, you can use the os.**remove()** function from the os module.

Examples

```
import os
#create a new file
with open("textfile.txt", "w") as f:
    f.write("Hello World")

#rename the file
os.rename("textfile.txt", "newfile.txt")

#delete the file
os.remove("newfile.txt")
```

Useful string methods for text file handling*

- The **strip()** method in Python is a string method that is used to remove leading and trailing whitespaces (spaces, tabs, and newline characters) from a string.
- If you only want to strip from the left (leading) or right (trailing) side, you can use the **lstrip()** or **rstrip()** methods, respectively.
- The **split()** method in Python is a string method used to split a string into a list of substrings based on a specified delimiter.

Examples

```
with open("fruits.txt", "r") as f:
    for lines in f:
        fruits = lines.strip("\n").strip(" ") #remove new line chars and spaces
        print(fruits)
        fruits = fruits.split(" ") #split fruits
        print(fruits)
        print()
```

```
apple orange
['apple', 'orange']

banana pear plum
['banana', 'pear', 'plum']

melon
['melon']
```

* See also Seminar 5.

Exception handling

Exception handling is a programming construct that allows you to manage and respond to errors that may occur during the execution of a program.

The syntax to handling exception in Python:

try:
block to contain the code that might raise an exception

except:
block to catch specific exceptions or handle general errors.

else:
block to execute code if no exceptions are raised in the try block. (Optional)

finally:
block to execute code that must run regardless of whether an exception occurred or not. (Optional)

Example:

```
import os
try:
  f = open("fruits.txt")
  print(f.read())
except Exception as e:
  print(e)
else:
  f.close()
finally:
  print("we continue if there is an error, if there is not")
```

```
[Errno 2] No such file or directory: 'fruits.txt'
we continue if there is an error, if there is not
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 11.

OOP in Python

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Class, constructor, data members and methods
- Destructor and garbage collector
- Instantiation (objects)
- Getters and setters, properties
- Method overloading
- Special attributes and methods
- Operator overloading
- Inheritance, method’s overriding, polymorphism
- Abstract methods
- Data hiding

Class, data members and methods

In object-oriented programming (OOP), a class is a blueprint or a template for creating objects. Class defines a set of data members (attributes, properties) and methods (functions, procedures).

Syntax of a class

```
class:
    #data members


    #methods
```

Neither the data members, not the methods part is mandatory

Example

```
class Circle:
    #data memebers
    pi_value = 3.14

    #methods
    def area(self, r):
        return r**2 * Circle.pi_value

    def perimeter(self, r):
      return 2 * r * Circle.pi_value
```

Constructor

Constructors are special methods and are generally used for instantiating an object.

Syntax of a constructor

```
def __init__(self*, other parameters):

    # body of the constructor
```

Example

```
class Circle:

    #constructor
    def __init__(self, radius):
        self._radius = radius
```

Constructors are used to …

- ❑ set the initial and/or default values of the attributes
- ❑ allocate the required resources
- ❑ perform the necessery setup

*Self parameter is a convention used in the method definitions of a class to refer to the instance of that class.

- ❑ It is the first parameter in the method
- ❑ It represents the instance on which the method is called.
- ❑ The name "self" is just a convention

- Creating a constructor is not mandatory, but highly recommended
- Unlike other object-oriented languages, a Python class can contain only one constructor
- Python does not have constructors with access modifiers such as private, public or protected

Destructor and garbage collector

A destructor is a special method in OOP that is responsible for releasing resources, performing cleanup, or executing finalization code when an object is no longer needed. Python uses automatic memory management with a garbage collector to reclaim memory occupied by objects that are no longer referenced.

The syntax of a destructor:

```
def __del__(self):

    #statements
```

- Destructors are raraly used in Python.
- We can control the setup and cleanup operations using the with statement

Example

```
class Number:

    #constructor
    def __init__(self, value):
        self._value = value

    def get_value(self):
        return self._value

    #destructor
    def __del__(self):
        print("the number does not exist anymore")

n = Number(8)
print("The value of the number: ", n.get_value())
```

```
the number does not exist anymore
The value of the number:  8
```

Object instantiation

Object instantiation refers to the process of creating an instance or an occurrence of a class, which is an object.

Syntax of instantiation (two ways:

Instance_variable_name =

ClassName(parameter_value1, parameter_value2, ...**)**

Instance_variable_name =

ClassName(parameter_name1 = parameter_value1, parameter_name2 = parameter_value2, ...**)**

Examples

```
class Circle:
    #data memebers
    pi_value = 3.14

    #methods
    def __init__(self, r):
        self.radius = r

    def area(self):
       return self.radius**2 * Circle.pi_value

#instantiation
c1 = Circle(5.0)  #a circle object with radius 5.0
c2 = Circle(3.0)  #a circle object with radius 3.0
```

Getters and setters, properties

In OOP, private data can not be directly accessed, only indirectly using getters, setters or properties.

- **Getters** are the methods used in Object-Oriented Programming (OOPS) which helps to access the private attributes from a class.
- **Setters** are the methods used in OOPS feature which helps to set the value to private attributes in a class.
- **Properties** allow you to define custom behavior when getting, setting, or deleting the value of an attribute.

Examples

```
class Circle:
    #constructor
    def __init__(self, radius):
        self._radius = radius

    #set method (setter)
    def set_radius(self, value):
        self._radius = value

    #get method (getter)
    def get_radius(self):
        return self._radius

c = Circle(6.0)
c._radius = 2.0 #this is not allowed!
c.set_radius(2.0) #using a setter instead
```

Getters and setters, properties II.

With the help of properties, class data can be accessed like doing it directly.
In Python, there are two ways to define a property.

Using **property() function**

```
def get_value:
    #statements

def set_value:
    #statements

def del_value:
    #statements

value = property(get_value, set_value, del_value*)
```

*del_value is optional

Using @**property** decorator

```
@property
def value(self):
    #statements

@value.setter
def value(self, new_value):
    #statements

@value.deleter   #this is optional
def value(self):
    #statements
```

CORVINUS UNIVERSITY of BUDAPEST

Example of properties

```
class Circle:

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        self._radius = value

c = Circle(4.2)
print(f"the original radius: {c.radius}")
c.radius = 3.1
print(f"the new radius: {c.radius}")
```

```
the original radius: 4.2
the new radius: 3.1
```

Static methods

Static methods are methods that belong to a class rather than an instance of the class. They don't have access to the instance or its attributes, and they are defined using the @staticmethod decorator above the method definition.

Syntax:

```
@staticmethod
def method_name(parameters):

        #statements
```

Example

```
class MyMath:

    @staticmethod
    def add_numbers(x, y):
        return x + y


a = 20
b = 30
print(f"{a} + {b} = {MyMath.add_numbers(a, b)}")
```

```
20 + 30 = 50
```

Method overloading

Method overloading refers to the ability to define multiple methods in a class with the same name but different parameter lists. Python does not support method overloading. However, Python provides two ways to achieve similar functionality: using default or arbitrary parameters (see seminar 9)

Ways of method overloading:

def method_name(**self,** param1 [= default_value1], param2 [= default_value2]…):

#statements

def method_name(**self, *args):**

#statements

Example1 (default parameters):

```
class Vector2D:

    def __init__(self, x = 0, y = 0): #two default parameters
        self._x = x
        self._y = y

    def getx(self):
        return self._x

    def gety(self):
        return self._y

u = Vector2D();
v = Vector2D(4)
z = Vector2D(5, 1)
print(f"u = ({u.getx()}, {u.gety()})")
print(f"v = ({v.getx()}, {v.gety()})")
print(f"z = ({z.getx()}, {z.gety()})")
```

```
u = (0, 0)
v = (4, 0)
z = (5, 1)
```

Method overloading

Method overloading refers to the ability to define multiple methods in a class with the same name but different parameter lists. Python does not support method overloading. However, Python provides two ways to achieve similar functionality: using default or arbitrary parameters (see seminar 9)

```
class Person:

    def __init__(self, name, *args):
        self._name = name
        s = ''
        for a in args:
            s = s + " " + a
        self._subjects = s

    def get_name(self):
        return self._name

    def get_subjects(self):
        l = []
        for s in self._subjects.split():
            l.append(s)
        return l

p = Person("Emma", "maths")
q = Person("Alice", "maths", "biology", "history")
print("The first person: ", p.get_name(), p.get_subjects())
print("The second person: ", q.get_name(), q.get_subjects())
```

```
The first person:  Emma ['maths']
The second person:  Alice ['maths', 'biology', 'history']
```

Example2:
arbitrary parameters

Special methods

Special methods are special functions in Python classes that are surrounded by double underscores on both sides of their name.

These methods provide a way for classes to define how they behave in various contexts, such as arithmetic operations (see next slide), comparisons (see next slide), and conversions.

Examples:

- __init__() the constructor
- __str__() string representation of an object called by print() and str(), gives a human-readable form
- __repr__() string representation of an object called by repr(), gives an unambigous form for the developer
- __len__() returns the length of an object, called by len() function
- __contains__() implements membership test operators such as in and not in

Operator overloading

Operator overloading in Python refers to the ability to define and redefine the behavior of operators for user-defined objects.

Special methods for operator overloading

- ❑ __add__() + operator
- ❑ __sub__() - operator
- ❑ __mul__() * operator
- ❑ __truediv__() / operator
- ❑ __mod__() % operator
- ❑ __pow__() ** operator
- ❑ __eq__() == operator
- ❑ __ne__() != operator
- ❑ __lt__() < operator
- ❑ __gt__() > operator

Examples

```
class Vector2D:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __add__(self, other):
        return Vector2D(self._x + other._x, self._y + other._y)

    def getx(self):
        return self._x

    def gety(self):
        return self._y

u = Vector2D(2, 5)
v = Vector2D(3, 1)
z = u + v
print(f"u + v = ({z.getx()},{z.gety()})")
```

```
u + v = (5,6)
```

Special attributes

Special attributes in Python are surrounded by double underscores on both sides of their name. These attributes have specific meaning.

Examples:

- __name__ name of the current modul or class
- __doc__ the docstring of the function, class or module
- __class__ returns the name of the class to wich an object belongs
- __module__ returns the name of the module in wich a class or function was defined
- __file__ provides the path to the file from which a module, class, or function was loaded.
- __bases__ returns a tuple containing the base classes of a class

CORVINUS UNIVERSITY of BUDAPEST

Inheritance

Inheritance allows a new class (called the derived or child class) to inherit attributes and methods from an existing class (called the base or parent class)

The syntax of inheritance:

```
class child_class_name(parent_class_name):
    #data members and methods
```

A class can inherit from more than one parent class.

Example:

```
class Number:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value

class Prime_number(Number):
    def __init__(self, value):
      super().__init__(value)
      if not self.is_prime():
            raise ValueError(f"{value} is not a prime number.")

    def is_prime(self):
        i = 2
        while i < self.value and self.value % i != 0:
            i = i + 1
        return i == self.value

try:
    p = Prime_number(5)
    print(p.value)
except Exception as e:
   print(e)
```

Abstract methods

- Abstract methods serve as a blueprint for methods that must be implemented by any concrete (i.e., non-abstract) subclasses.
- Abstract classes themselves cannot be instantiated.
- In Python, to use abstract methods, needs to use the abc module.

Without using abc module, an abstract-like method:

```
def abstract_method_name():
        pass
```

Or:

```
def abstract_method_name():
        raise NotImplementedError("Subclasses
              must implement the area method.")
```

Example:

```
class Animal():

    def make_sound(self):
        pass

class Dog(Animal):

    def make_sound(self):
        return "Woof"

class Bird(Animal):

    def make_sound(self):
        return "Chirp"

d = Dog()
print("the dog says: ", {d.make_sound()})
```

```
the dog says:  {'Woof'}
```

Data hiding

In Python, unlike some other programming languages like Java, there are no strict keywords for defining access modifiers such as "private," "public," or "protected." Instead, Python uses naming conventions to indicate the intended visibility of attributes and methods.

The naming conventions:

- **Public:** Attributes and methods with no leading underscore
- **Protected:** Attributes and methods with a single leading underscore
- **Private:** Attributes and methods with a double leading underscore

Examples

```
class MyClass:

    def __init__(self):
        self.__private_attribute = 10
        self._protected_attribute = 30
        self.public_attribute = 50

    def __private_method(self):
        print("This is a private method.")

    def _private_method(self):
        print("This is a protected method.")

    def public_method(self):
        print("This is a public method.")
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 12.

Solving a complex problem in Python

CORVINUS UNIVERSITY of BUDAPEST

Agenda

- Caesar cipher
- Caesar decipher
- Break Caesar code
- Break code with frequency analysis
- Regular expressions in Python

Caesar cipher

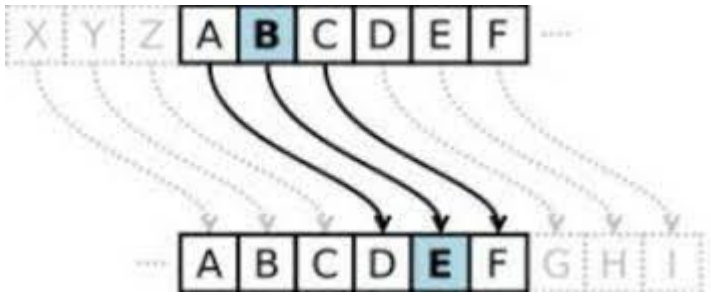
The Caesar cipher is a substitution cipher where each letter in the plaintext is shifted a certain number of places down or up the alphabet. This shift is known as the key.

Formula: $E_n x = (x + n) \% 26$

If shift = 3, then

- A becomes D
- B becomes E
- C becomes F

and so on



It is easy to break this code since there are only 25 possible key

Example:

```
1  abc_upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
2  shift = 3
3  message = 'HELLO WORLD' #use capital letters only
4
5  #encoding
6  encoded_message = ""
7  for c in message:
8    if c in abc_upper:
9      index = 0
10     while abc_upper[index] != c:
11         index = index + 1
12     c = abc_upper[(index + shift) % len(abc_upper)]
13   encoded_message += c
14 print(encoded_message)
15
```

```
KHOOR ZRUOG
```

Caesar decipher

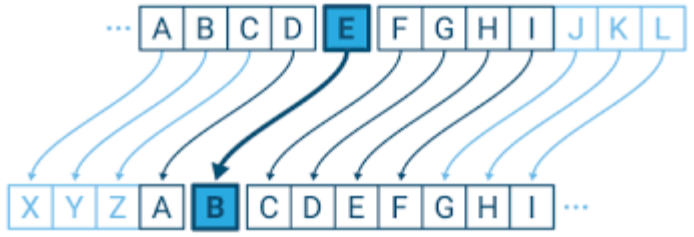
The Caesar decipher involves shifting the letters in the opposite direction of the original encryption.

Formula: $D_nx = (x - n) \% 26$

If shift = 3, then

- D becomes A
- E becomes B
- F becomes C

and so on



Example:

```
abc_upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
shift = 3
encoded_message = 'KHOOR ZRUOG' #use capital letters only

#decoding
decoded_message = ""
for c in encoded_message:
  if c in abc_upper:
    index = 0
    while abc_upper[index] != c:
        index = index + 1
    c = abc_upper[(index - shift) % 26]
  decoded_message += c
print(decoded_message)

```

```
HELLO WORLD
```

BreakCaesar cipher

To break Ceasar code, try all possible keys from zero to twenty five (brute-force attack)*.

Example:

```
abc_upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
shift = 3
encoded_message = 'KHOOR ZRUOG' #use capital letters only

#decoding with brute-force attack

shift = 1
ch = ''
while shift < 26 and ch != 'y':
 decoded_message = ""
 for c in encoded_message:
  if c in abc_upper:
    i = 0
    while shift < 26 and c != abc_upper[i]:
        i = i + 1
    c = abc_upper[(i - shift) % 26]
  decoded_message += c
 print(decoded_message, end = '\n')
 ch = input("is it OK? (y/n)")
 if ch == 'y':
        print('the text has been decrypted using shift ', shift)
 else:
    shift = shift + 1
print(decoded_message)
```

```
JGNNQ YQTNF
is it OK? (y/n)n
IFMMP XPSME
is it OK? (y/n)n
HELLO WORLD
is it OK? (y/n)y
the text has been decrypted using shift  3
HELLO WORLD
```

* Simplier solution: first determine the key, then use it at decryption

Break code with frequency analysis

Frequency analysis relies on the fact that certain elements in the plaintext will occur more frequently than others.

Steps

- ❑ Collect data and create frequency distribution table
- ❑ Compare the distributon with expected distribution coming from the practice or from a plaintext
- ❑ Identify common patterns
- ❑ Try it
- ❑ If needed, repeat these steps until the result is acceptable

Frequency of common english letters

Letter	Frequency	Letter	Frequency
e	12.7020%	m	2.4060%
t	9.0560%	w	2.3600%
a	8.1670%	f	2.2280%
o	7.5070%	g	2.0150%
i	6.9660%	y	1.9740%
n	6.7490%	p	1.9290%
s	6.3270%	b	1.4920%
h	6.0940%	v	0.9780%
r	5.9870%	k	0.7720%
d	4.2530%	j	0.1530%
l	4.0250%	x	0.1500%
c	2.7820%	q	0.0950%
u	2.7580%	z	0.0740%

Break code with frequency analysis

Ciphertext:

```
FKDSWHUPUVKHUORFNKROPHVPUVKHUORFNK
XFFHVVIXOHOGHUOBPHGLFDOPDQZHOOHVWH
DLGDVPXFKEHIRUHDQGLPXVWDGPLWWKDWKL
DOOEBQRPHDQVDOOLZRXOGVXJJHVWIRUHAD
RQWKHRFFDVLRQRIWKHFKDQJHLWFHUWDLQC
```

{'E': 23558, 'T': 17716, 'A': 15714, 'O': 15380, 'I': 13269, 'H': 12639,

Plaintext:

```
Sherlock Holmes took his bottle from the corner
his hypodermic syringe from its neat morocco ca
white, nervous fingers he adjusted the delicate
his left shirt-cuff.  For some little time his
upon the sinewy forearm and wrist all dotted an
innumerable puncture-marks. Finally he thrust t
pressed down the tiny piston, and sank back int
arm-chair with a long sigh of satisfaction.
```

{'H': 1044, 'W': 830, 'D': 813, 'R': 809, 'L': 779, 'V': 666}

```python
def sort_freq(self, text):
    d = {}
    for c in text:
        if c in d and c in Cypher._abc_upper:
            d[c] += 1
        else:
            d[c] = 1
    l = list(d.items())
    for i in range(len(l) - 1):
        for j in range(i + 1, len(l)):
            if l[j][1] > l[i][1]:
                l[i], l[j] = l[j], l[i]

    return dict(l)
```

Mapping letters:

{'H': 'E', 'W': 'T', 'D': 'A', 'R': 'O', 'L': 'I', 'V': 'H', 'Q': 'N', 'K': 'S', 'U': 'R', 'O': 'D', 'G': 'L', 'P': 'U', 'F': 'M', 'X': 'W', 'B': 'C', 'I': 'F', 'Z': 'Y', 'J': 'G', 'S': 'P', 'E': 'B', 'Y': 'V', 'N': 'K', 'A': 'J', 'M': 'X', 'T': 'Q', 'C': 'Z'}

CORVINUS UNIVERSITY of BUDAPEST

Regular expressions (see also http://www.pyregex.com/)

Regular expressions (regex) in Python are a powerful tool for pattern matching and text manipulation

Syntax of pattern:

- **Literal** means full match with the pattern
- **Character class** [ ] means matching with one out of the pattern's characters
- * means zero or more occurances of the preceding groups,
- + means one or more occurances of the preceding
- ? means zero or one occurance of the preceding character,
- . means any character except newline of the preceding character
- ^ means matching the start of the line
- $ means matcing the end of the line
- \ escapes a special character treated as literal (\t tab, \b word boundary, \s whitespace)
- | means logical OR
- ( ) means grouping and capturing

Regular expressions

Example:

```
import re
text = """BookID	BookTitle	BookAuthor	BookPrice
1	Iron Flame (The Empyrean, 2)	Rebecca Yarros	18.42
2	The Woman in Me	Britney Spears	20.93
3	My Name Is Barbra	Barbra Streisand	31.máj
4	Friends, Lovers, and the Big Terrible Thing: A Memoir	Matthew Perry	23.99
5	How to Catch a Turkey	Adam Wallace	5.65
6	Fourth Wing (The Empyrean, 1)	Rebecca Yarros	16.99
7	Unwoke: How to Defeat Cultural Marxism in America	Unknown	27.43
"""
pattern = re.compile(r'\t(.*?)\t') #match the title of the books
matches = pattern.findall(text)
print(matches, '\n')
#match rows containing Britney Spears
pattern = re.compile(r'^.*\bBritney Spears\b.*$', flags = re.MULTILINE)
matches = pattern.findall(text)
print(matches)
```

```
['BookTitle', 'Iron Flame (The Empyrean, 2)', 'The Woman in Me', 'My Name Is Barbra', 'Friends, Lovers, and the Big Terrible Th
ing: A Memoir', 'How to Catch a Turkey', 'Fourth Wing (The Empyrean, 1)', 'Unwoke: How to Defeat Cultural Marxism in America']

['2\tThe Woman in Me\tBritney Spears\t20.93']
```

CORVINUS UNIVERSITY of BUDAPEST

Python seminar 13.

Summary

CORVINUS UNIVERSITY of BUDAPEST

What did we learn during the semester?

- Simple data types (int, float, bool, str)
- Conditional branches (if – elif –else, match-case, ternary operator)
- Loops (for loop over a range, container, file, while loop)
- Compound data types (list, tuple, set, dictionary, text file)
- Basic algorithms (decision, summarization … bubble-sort)
- Algorithm description tools (flowchart, structogram, pseudocode)
- Structuring the program (functions, procedures, modules)
- Object-oriented programming (class, data members and methods …)

Programming = data structures + algorithms

Programming ≠ Python syntax

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

f-strings (using variables in strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = f"{first_name} {last_name}"
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print(f"The alien's color is {alien['color']}")
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(f"{name} loves {number}")
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(f"{name} loves a number")
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(f"{number} is a favorite")
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print(f"Hello, {name}!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

CORVINUS UNIVERSITY of BUDAPEST

Final assignment

- **First part** (~ 25 min) – short tasks
- **Second part** (~ 45 min) – complex task
- Main topics:
    - Application of basic algorithms
    - Implement algorithm described with a flowchart, structogram or pseudocode
    - Working with simple and complex data structures (text files are included)
    - Creating simple functions and/or procedures
    - *OOP will not be included!*

Presentation

- After the final assignment
- One presenter per group is enough
- Presentation time: 5-6 min
- Discussion, evaluation: 3-4 min