Part 6

Intro to Programming

10.10.2024

Last Week

- mixed types of items in lists
- matrices
- references
- lists as arguments and return values
- dictionary
- tuple

Reading files

Typical action in programs

Using text files is easy in Python

Other file formats may be more complicated

Example

Reading file contents

with-block handles automatic closing of file

```
with open("example.txt") as new_file:
    contents = new_file.read()
    print(contents)
```

Handling files one row at a time

Method **read** reads the contents of the entire file

Often, it's easier to read file one row at a time

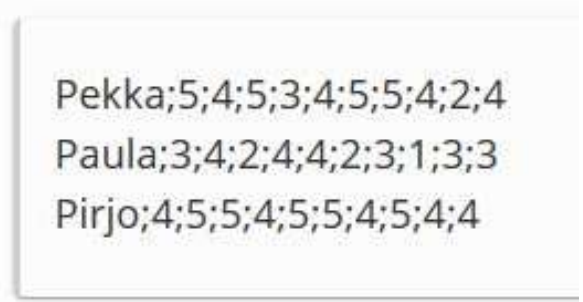
This is easy with the for statement

```
with open("example.txt") as new_file:
    count = 0
    total_length = 0

    for line in new_file:
        line = line.replace("\n", "")
        count += 1
        print("Line", count, line)
        length = len(line)
        total_length += length

print("Total length of lines:", total_length)
```

CSV Files

Comma Separated Values

Several data points in one row, separated with suitable character (i.e. a *separator*)



Splitting strings

String can be splitted into a list with the **split** method

The separator character is provided as an argument

```
text = "monkey,banana,harpsichord"
words = text.split(",")
for word in words:
    print(word)
```

Reading same file several times

File can be read in **with-**block only once

```
with open("people.csv") as new_file:
    # print out the names
    for line in new_file:
        parts = line.split(";")
        print("Name:", parts[0])

    # find the oldest
    age_of_oldest = -1
    for line in new_file:
        parts = line.split(";")
        name = parts[0]
        age = int(parts[1])
        if age > age_of_oldest:
            age_of_oldest = age
            oldest = name
    print("the oldest is", oldest)
```

Removing extra characters

Reading data from file also returns line feed characters

Easiest way to get rid of these is to use the **strip** method

```
>>> " tryout ".strip()
'tryout'
>>> "\n\ntest\n".strip()
'test'
>>>
```

Creating a file

New file can be created by providing character 'w' as the second argument for the open function

```
with open("new_file.txt", "w") as my_file:
    # code to write something to the file
```

Note, that if file exists, is will be overwritten!

Writing to file

Lines can be written to the file with the **write** method

Please note, that the linefeeds must be added manually

```
with open("new_file.txt", "w") as my_file:
    my_file.write("Hello there!\n")
    my_file.write("This is the second line\n")
    my_file.write("This is the last line\n")
```

Appending data to file

New data can be appended after the existing data in the 'a' file mode

```
with open("new_file.txt", "a") as my_file:
    my_file.write("This is the 4th line\n")
    my_file.write("And yet another line.\n")
```

Writing a CSV file

```
with open("coders.csv", "w") as my_file:
    for coder in coders:
        line = f"{coder[0]};{coder[1]};{coder[2]};{coder[3]}"
        my_file.write(line+"\n")
```

```
with open("coders.csv", "w") as my_file:
    for coder in coders:
        line = ""
        for value in coder:
            line += f"{value};"
        line = line[:-1]
        my_file.write(line+"\n")
```

Emptying and removing files

```
open('file_to_be_cleared.txt', 'w').close()
```

```
# the command to delete files is in the os module
import os

os.remove("unnecessary_file.csv")
```

Validating inputs

Typical examples of erroneous input data:

- **missing or empty input values** in mandatory fields, such as empty strings when the length of the string is critical
- **negative values** where only positive values are accepted, such as -15 as the amount of an ingredient in a recipe
- **missing files** or typos in filenames
- values that **are too small or too large**, for example when working with dates and times
- **invalid indexes,** such as trying to access index 3 in the string "hey"
- values of a **wrong type,** such as strings when integers are expected

Validating input (2)

It's often possible to prepare for erroneous data

Inputted data can be validate with conditions

```
age = int(input("Please type in your age: "))
if age >= 0 and age <= 150:
    print("That is a fine age")
else:
    print("This is not a valid age")
```

Exceptions

Some errors are difficult to check with conditional statements only

Exceptions are errors that occur during the execution of programs

Preparing for exceptions

Exceptions can be caught with try-except structure

```
try:
    age = int(input("Please type in your age: "))
except ValueError:
    age = -1

if age >= 0 and age <= 150:
    print("That is a fine age")
else:
    print("This is not a valid age")
```

Typical exceptions

ValueError

TypeError

IndexError

ZeroDivisionError

Exceptions in file handling

More than one except block

If more than one error may be thrown in the **try** block, it can be followed with more than one **except** block

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except FileNotFoundError:
    print("The file example.txt was not found")
except PermissionError:
    print("No permission to access the file example.txt")
```

Undefined exception

All possible exceptions can be caught by not defining the exception in the except block

Note, that this also catches the errors made by the programmer

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except:
    print("There was an error when reading the file.")
```

Passing exceptions

Exception thrown in a function is passed back to the caller of the function

```
def testing(x):
    print(int(x) + 1)

try:
    number = input("Please type in a number: ")
    testing(number)
except:
    print("Something went wrong")
```

Throwing exceptions

Exception can be thrown manually with keyword **raise**

This is a good method for handling e.g. invalid parameters in functions

```
def factorial(n):
    if n < 0:
        raise ValueError("The input was negative: " + str(n))
    k = 1
    for i in range(2, n + 1):
        k *= i
    return k
```

Local variables

Variables defined in functions are only visible inside functions

```
def testing():
    x = 5
    print(x)

testing()
print(x)
```

```
5
NameError: name 'x' is not defined
```

Keyword global

Variables defined in the main program can be used inside functions with keyword **global**

This is, in general, a bad idea. Instead of side effects, the functions should return a value

```
def testing():
    global x
    x = 3
    print(x)

x = 5
testing()
print(x)
```

Next week

Modules

Random

Handling times and dates

More file handling

Own modules

More about Python's features

Few words about the exam