מג'אווה לפייתון-קבצים וחריגות

טיפול בקבצים-

גם בפייתון כמו ג'אווה ניתן לפתוח קובץ קיים ולכתוב אליו\ לקרוא ממנו , או לייצר אחד חדש.
בג'אווה כשרוצים להשתמש בקובץ צריך לייבא ספריה מיוחדת ,להפעיל פונקציה על הקובץ ולשמור את הנתונים באובייקט מטיפוס File.
עם פייתון לרוב החיים פשוטים יותר ,וכשזה מגיע לטיפול בקבצים אפשר להחשיב את פייתון כ-"הוואי של שפות התכנות" (אני פשוט מניח שהחיים בהוואי יותר טובים, בכל מקרה תוכלו לקחת את המשל לכל מקום שנראה לכם כ-"חיים הטובים").
פייתון מתייחס אחרת לקבצי טקסט ולקבצים בינאריים.
כל שורה של קוד כוללת רצף של תווים והם יוצרים קובץ טקסט, וכל שורה בקובץ נגמרת עם תו מיוחד שנקרא EOL (End Of Line) כמו ירידת שורה או פסיק, והתו מסמן למפרש מתי נגמרת השורה ומתחילה שורה חדשה בקובץ.
בשביל לפתוח קובץ משתמשים בפונקציה בנויה מראש " open()" שמקבלת כפרמטר את שם המסמך (ה-path שלו), ותו הרשאה: r- לקריאה(reading); w- לכתיבה מחדש של המסמך, כלומר דריסה של התוכן הקיים(writing); a- הוספה לקובץ הקיים (appending) ו- r+ -גם לקרוא וגם לכתוב.
הפונקציה מחזירה אובייקט מטיפוס קלט לקובץ טקסט:

```
>>> file = open("stam.txt","r")
>>> type(file)
<class '_io.TextIOWrapper'>
```

התו הרשאה לא מחייב וכברירת מחדל הוא נחשב 'r' .

קריאה-

יש כמה דרכים לקרוא ממקובץ, הדרך האינטואיטיבית היא לקרוא שורה אחר שורה מהקובץ בלולאה, אך ניתן גם לקבל את כל תוכן הקובץ כמחרוזת עם הפונקציה read() של קבצים שמקבלת כפרמטר int עם כמות התווים שנרצה לקרוא מהקובץ, אך אם לא נתנו לה ארגומנט היא תקרא את כל המסמך:

```
>>> file = open("stam.txt","r")
>>> print(file.read())
```

הקריאה של הקובץ היא בצורה של חוצץ בזיכרון ומצביע לראש השורה, כך שכל שורה שנקראת המצביע עומד על ראש כל תו שנקרא ועובר לשורה הבאה בסוף כשהוא מגיע לEOL ומוחק מהחוצץ בזיכרון את השורה שהוא קרא הרגע.
בסוף קריאת המסמך המצביע עומד על סוף המסמך.
הערה: אי אפשר לקרוא את המסמך שוב, אלא אם נפתח אותו מחדש, כי הקריאה נעשית בהתאם למצביע

כתיבה-

בדומה לפונקציה read() יש גם פונקציה write() שמאפשרת לכתוב למסמך.
כתיבה עם הפונקציה write דורסת את מה שהיה כתוב לפני בטקסט (אם היה) ומחזירה את מספר התווים שהכנסנו.
לאחר השימוש בפונקציה write() יש לסגור את השימוש האובייקט "קובץ" כלומר להראות לתוכנה שאין לנו כוונה שנות אותו יותר.
אפשר גם לכתוב מסמך חדש במידה והוא לא קיים והוא ישמר בשם ומיקום שהכנסנו לפונקציה, כברירת מחדל באותה תיקייה של התוכנית:

```
>>> file = open('new_file.txt','w')
>>> file.write("This is the first line I write in this file")
43
>>> file.write("\nThis is the second line I write in this file")
45
>>> file.close()
>>> file = open('new_file.txt','r')
>>> print(file.read())
This is the first line I write in this file
This is the second line I write in this file
>>>
```

בדומה להרשאת כתיבה, ניתן גם להוסיף תוכן לקובץ הקיים עם ההרשאה 'a', והיא מאפשרת להוסיף טקסט לסוף הקובץ:

```
file = open('new_file.txt','a')
file.write("\nThis is a new line I just added")
file.close()
file= open('new_file.txt')
print(file.read())
This is the first line I write in this file
This is the second line I write in this file
This is a new line I just added
```

אם נבחר להשתמש ב-r+ זה יאפשר לנו גם לקרוא וגם לכתוב לקובץ, אך הכתיבה תעשה למקום בו המצביע עומד בדיוק, מה שעלול לדרוס שורות מתוך הטקסט.

המילה השמורה with-

אם עד עכשיו לא הרגשתם בהוואי חכו שתשמעו על הפונקציה הבאה:
השימוש בopen() ו-close() מועד לשכחה , לכן המפתחים של פייתון סיפקו דרך שבה לא נצטרך להיכנס בכלל לתסבוכת הזאת.
במקום להגדיר פתיחה וסגירה של קובץ, ישנה אפשרות להגדיר בלוק שבסופו יתבצע סגירה אוטומטית של הקובץ, עם המילה שמורה with, פתיחת קובץ והגדרת שם הקובץ עם המילה שמורה as:

```
>>> with open("hawaii.txt",'r+') as file:
...  print(file.read())
...  file.write("\nPYTHON==GOOD LIFE")
  ,---,
,--.' |                                          ,--,    ,--,
|  |  :                       .---.           ,--.'|  ,--.'|
:  :  :                      /. ./|           |  |,   |  |,
:  |  |,--.  ,--.--.       .-'-. ' |  ,--.--.   `--'_   `--'_
|  :  '   | /       \     /___/ \: | /       \  ,' ,'|  ,' ,'|
|  |   /' :.--.  .-. | .-'..'    ' ..--.  .-. |  '  | |  '  | |
'  :  | | | \__\/: . ./___/ \:     ' \__\/: . . |  | :  |  | :
|  |  ' | : ," .--.; |.    \  ' .\    ," .--.; |  '  : |__'  : |__
|  :  :_:,'/  /  ,.  | \    \   ' \ |/  /  ,.  | |  | '.'|  | '.'|
|  | ,'    ;  :    .'   \ \    \  |--";  :    .'   \;  :    ;  :    ;
`--''      |  ,     .-./  \    \ |   |  ,     .-./|  ,   /|  ,   /
            `--`---'        '---"     `--`---'     ---`-'  ---`-'
19
```

טיפול בשגיאות-

תכניות נגמרות ברגע שהן נתקלות בשגיאה במהלך ריצתן.
תמיד השאיפה היא להימנע כמה שאפשר משגיאות, אך לפעמים זה לא בידי המתכנתים.
במקרים כאלה לא נרצה שהתוכנית תפסיק לחלוטין באמצע ריצתה, במיוחד כשהשגיאה לא קשורה להמשך התוכנית.
בפייתון יש שני סוגים של שגיאות- שגיאות סינטקס, וחריגות.
שגיאות סינטקס קורות כאשר המפרש מבחין בביטוי או הכרזה שאינם נכונים מבחינת צורת הכתיבה שלהם:

```
>>> print( 0 / 0 ))
File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

שימו לב שהשלב של זיהוי שגיאת הסינטקס מגיע עוד לפני השלב שהמפרש מנסה להעריך את הביטוי, למשל במקרה לעיל המפרש מסמן על שגיאה תחבירית בחלק של הסוגרים הכפולים, ולא על החילוק באפס.
אם היינו מתקנים את שגיאת הסינטקס היינו נתקלים בשגיאה מסוג אחר:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

הפעם "נזרקת" שגיאת חריגה של חלוקה באפס.
סוג זה של שגיאה קורה כשהקוד כתוב נכון מבחינה תחבירית, אך עדיין המפרש נתקל בכשל במהלך הערכה של הביטוי.

זריקת שגיאות מבוקרות-
ניתן לזרוק שגיאה בעצמנו עם המילה השמורה raise ואחריה ליצור אובייקט מסוג חריגה עם פרמטר מטיפוס מחרוזת שמתאר את סוג החריגה :

```
>>> def error_fun(x):
...  raise Exception("Invalid input")
...
>>> error_fun(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in error_fun
Exception: Invalid input
```

אפשר גם להשתמש בפונקציה assert() כדי לוודא שהמכשיר שמריץ את התוכנית יכול לעמוד בקריטריונים שלה, או שנרצה לבחון את הקוד שלנו לפני תחילת התוכנית כדי לוודא שהתוכנית לא תיעצר באמצע ריצתה, או שחלילה תפגע במכשיר שמריץ אותה.
עם הפונקציה assert() אנחנו בודקים שתנאי כלשהו מתקיים, ובמידה ולא אז הפונקציה תזרוק שגיאה מכוונת עוד בהתחלה ותמנע מהתוכנית ליפול בהמשך:

```
import sys #for sys.platform
assert ('linux' in sys.platform), "This code runs on Linux only."
```

בדוגמא לעיל אם מערכת ההפעלה של המכשיר שהריץ את הקוד היא לינוקס הפונקציה לא תזרוק שגיאה, אחרת היא תזרוק שגיאת assert שהסיבה לה מופיעה במחרוזת שהגדרנו אחרי ה-',' .

try ו- except -

try ו-except , שבג'אווה מוכרים כ-try ו- catch , הוא מנגנון שמאפשר לנסות לבצע רצף של פקודות ולהגדיר מה יקרה אם נזרקה חריגה במהלך ריצת הפקודות.
עם המילה השמורה try אנו מורים לתוכנית להריץ את הבלוק שיופיע מתחתיה , ועם except מורים לבצע את רצף הפקודות שמופיע מתחת ל-except רק במקרה שנתפסה שגיאה בזמן ריצת הבלוק של ה-try.

```
try:
    10/x
except:
    pass
```

בפייתון לא ניתן לכתוב בלוק ריק כמו בג'אווה, לכן משתמשים במילה השמורה pass כשאין פקודות לביצוע.
אם הערך של x היה 0 הייתה אמורה להתקבל שגיאת חלוקה באפס, השגיאה אכן תגיעה אך היא תתפסה ב-except שיבצע את הפקודה שבאה בבלוק שלו במקום :

```
>>> def devi(x):
...  try:
...   10/x
...  except:
...   print(f'EXCEPTION')
...
>>> devi(10)
>>> devi(0)
EXCEPTION
```

מה שאנחנו לא זוכים לראות זה את סוג השגיאה שנתפסה, בשביל לראות את סוג השגיאה נצרך להגדיר לexcept מה סוג השגיאה שעלולה לקרות:

```
>>> def devi(x):
...  try:
...   10/x
...  except ZeroDivisionError as error:
...   print(error)
...
>>> devi(0)
division by zero
```

לא מומלץ לתפוס שגיאות מבלי להגדיר אותן מראש מעוד סיבה- ההגדרה של except מבלי להגדיר את סוג החריגה תתפוס את **כל** סוגי החריגות, ולרוב לא נרצה זאת.
הדוגמא הבאה תמחיש את זה:

```
while True:
  try:
   print("I am in a loopppppppp ☺" )
  except:
   print("Hooo I'm still in the loop ☹")
```

גם אם ננסה לעצור את הלולאה האינסופית הזאת עם ctrl+c בטרמינל, נראה שהיא לא תיעצר מהסיבה הפשוטה שהיא תופסת גם interruptions, כלומר היא תופסת את כל סוגי החריגות גם אלה אינן שגיאות של התוכנית.
אבל אם במקום נגדיר את סוג החריגה שעלולה להתרחש העצירה תתבצע כמתוכנן:

```
while True:
  try:
   print("I am in a loopppppppp ☺" )
  except Exception:
   print("finally I stopped!!! ")
```

בקשר לחריגה מטיפוס Exception: האובייקט הוא טיפוס אב לרוב סוגי החריגות הקיימות בשפה למעת חריגות BaseException, systemExit, KeyboardInterrupt ו- GeneratorExit שהן חריגות מיוחדות.
במקרים בהם לא ידוע לנו סוג החריגה שעלולה לקרות הוא יכול לשמש כגלגל הצלה, אך תמיד כדאי להשתמש בסוג החריגה המקורי כדי: א. לידע את שאר המתכנתים על סוג שגאיות שעלולות לקרות, ב. כדי להגדיר התנהגות ספציפית במקרה של סוג מסוים של חריגה:

```
>>> def file_reader(file_path):
...  try:
...   with open(file_path) as file:
...    file_str = file.read()
...    10/len(file_str)
...  except FileNotFoundError as fnt:
...   print("there is no such file, try again…")
...  except ZeroDivisionError:
...   print("Are you crazy? Did you just tried to divide by 0?!")
>>> file_reader("not_such_file.txt")
there is no such file, try again…
>>> file_reader("empty.txt")
Are you crazy? Did you just tried to divied by 0?!
```

כשחריגות מופיעות על המסך ועוצרות את ריצת התוכנית הן מציגות את מקור השגיאה, כלומר מאיזו קריאה בתוכנית (מאיזו שורה בקוד) היא נזרקה.
כשאנחנו תופסים חריגות בexcept נעדיף שזה יציג לנו את מקור החריגה ולא רק את שמה, כדי לעשות את זה נוכל להשתמש בtaceback" module" שמאפשר לנו בקלות לקבל מידע על החריגה:

```
>>> import traceback
>>> try:
...  raise Exception("I am an Exception >:-) ")
... except Exception as e:
...  print("just printing the exception")
...  print(e,'\n')
...  print("printing the traceback")
...  traceback.print_exc()
...  print("\nprinting traceback.format_exc():")
...  print(traceback.format_exc())
...

just printing the exception
I am an Exception >:-)

printing the traceback
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: I am an Exception >:-)

printing traceback.format_exc():
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
Exception: I am an Exception >:-)
```

וכמו בלולאות גם לtry ו- except יש else, רק שבמקרה זה הבלוק שלו מתבצע רק **אם לא** נתפסו חריגות ב-except.
ולפעמים נרצה שתתבצע פעולת "ניקוי" גם אם זרקה חריגה וגם אם לא, למשל פתחנו מסמך ונרצה לסגור אותו בין אם הייתה חריגה ובין אם לאו, לא נוכל לסגור את המסמך בבלוק של ה-try כי יכול להיות שלא נגיע לשורת קוד הזו וישר נכנס לבלוק של ה-except , במקרה זה נוכל להשתמש ב-finally שמגדיר בלוק שיקרה גם בין אם נתפסה חריגה ובין אם לא:

```
>>> def file_writer(file_path):
...  try:
...   file = open(file_path)
...   file_str = file.read()
...   10/len(file_str)
...  except FileNotFoundError as fnt:
...   print("there is no such file, try again.")
...  except ZeroDivisionError:
...   print("Are you crazy? Did you just tried to divide by 0?!")
...  else:
...   file.write("No exceptions were cought")
...  finally:
...   try:
...    file.close()
...   except NameError:
...    pass
...
>>> file_writer("not_such_file.txt")
there is no such file, try again.
>>> file_writer("empty.txt")
Are you crazy? Did you just tried to divide by 0?!
```