PHOT 110: Introduction to programming

LECTURE 06

Michaël Barbier, Spring semester (2023-2024)

HANDLING RUNTIME ERRORS (TEXTBOOK CH. 4)

DIFFERENT KINDS OF ERRORS

Remember that we have different types of errors

- Syntax errors: code inconsistent with the Python language, the code will not run (i.e. not start).
- Runtime errors: exceptions occurring while the program runs, the code will crash.
- Semantic errors: the code does not what was intended

Try to make the following files error free:

- `lecture_11_ex_errors_books.py`
- `lecture_11_ex_errors_runtime.py`

EXAMPLE: INTEGER DIVISION

Remember the function for division:

```python
def div(number, divisor):
    """ divides n/m and returns the quotient and rest """
    quotient = number // divisor
    remainder = number % divisor

    return quotient, remainder

print("Division of n/m:")
n = int(input("\tn = "))
m = int(input("\tm = "))
quotient, remainder = div(n, m)
print(f"Dividing {n} by {m} = {quotient}, rest = {remaind
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```
# input function returns a string
# n_str = input("n = ")
n_str = "5.6"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
ValueError: invalid literal for int() with base 10: '5.6'
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
# input function returns a string
# n_str = input("n = ")
n_str = "4,1"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
ValueError: invalid literal for int() with base 10: '4,1'
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
# input function returns a string
# n_str = input("n = ")
n_str = "4.00"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
ValueError: invalid literal for int() with base 10: '4.00'
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
# input function returns a string
# n_str = input("n = ")
n_str = "0x002A"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
ValueError: invalid literal for int() with base 10: '0x002A'
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
# input function returns a string
# n_str = input("n = ")
n_str = "010"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
Provided number n = 10 with type = <class 'int'>
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
# input function returns a string
# n_str = input("n = ")
n_str = "5-2"
n = int(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
```

```
ValueError: invalid literal for int() with base 10: '5-2'
```

USER INPUT AND RUNTIME ERRORS

- User input can be anything → runtime errors / crashes
  - runtime errors
  - undefined behavior: no crash but wrong results

```python
print(f"The division {-9} / {4} = {-9 // 4} with remainde
print(f"The division {9} / {-4} = {9 // -4} with remainde
```

```
The division -9 / 4 = -3 with remainder = 3
The division 9 / -4 = -3 with remainder = -3
```

USER INPUT AND RUNTIME ERRORS

- Two options to prevent runtime error crashes:
  1. Anticipate any error: **Validate** the input data
  2. **Handle** the error when it occurs

Try to prevent the error by input validation?
Or handle the error “when it already occurred”?

In Python preventing a problem is not always better than treating it !

OPTION (1): TEST INPUT ON VALIDITY

- We can use `eval` function to evaluate a string expression
- function `isinstance(object, type)` checks if an object is of type

```python
# n_str = input("provide a number for n = ")
n_str = "1.3"
n = eval(n_str)
print(f"Provided number n = {n} with type = {type(n)}")
# return validity
if not isinstance(n, int):
  print("Input is invalid !")
```

```
Provided number n = 1.3 with type = <class 'float'>
Input is invalid !
```

OPTION (2): ERROR HANDLING

- When a runtime error occurs Python raises an exception
- To catch an error we use the `try` - `except` block
    - Statements in the `try` block are executed until an exception is encountered
    - The `except` block is executed on encountering an exception

```
try:
  <statements>
except:
  <statements>
```

OPTION (2): ERROR HANDLING

- When a runtime error occurs Python raises an exception
- To catch an error we use the `try` - `except` block
  - Statements in the `try` block are executed until an exception is encountered
  - The `except` block is executed on encountering an exception

```python
# n_str = input("provide a number for n = ")
try:
  n_str = "1.3"
  n = int(n_str)
except:
  print("Input is invalid !")
```

```
Input is invalid !
```

RECOVERING FROM RUNTIME ERRORS

- If the user input was invalid:
    - Allow the user to **try again**
    - **Specify the problem** to the user
- Look at the Python script file: `lecture_11_ex_errors_validate_input.py`

MORE COMPLEX ERROR-HANDLING

```python
try:
  n = int("3"); m = int("5.5")
  quotient = n // m
except ZeroDivisionError:
  print("Can't divide by zero")
except ValueError:
  print("Please provide two integers")
else:
  print(f"quotient = {quotient}")
finally:
  # Always executed
  print(f"You tried to divide n / m")
```

```
Please provide two integers
You tried to divide n / m
```

MORE COMPLEX ERROR-HANDLING

```python
try:
  n = int("5"); m = int("0")
  quotient = n // m
except ZeroDivisionError:
  print("Can't divide by zero")
except ValueError:
  print("Please provide two integers")
else:
  print(f"quotient = {quotient}")
finally:
  # Always executed
  print(f"You tried to divide n / m")
```

```
Can't divide by zero
You tried to divide n / m
```

MORE COMPLEX ERROR-HANDLING

```python
try:
  n = int("6"); m = int("2")
  quotient = n // m
except ZeroDivisionError:
  print("Can't divide by zero")
except ValueError:
  print("Please provide two integers")
else:
  print(f"quotient = {quotient}")
finally:
  # Always executed
  print(f"You tried to divide n / m")
```

```
quotient = 3
You tried to divide n / m
```

MORE COMPLEX ERROR-HANDLING: `else`

- `else` keyword allows to split the error handling of the `try` block:
  - `try`-part which you want to catch errors **now**
  - `else`-part which you have code that has its own error handling, or should crash if a problem occurs.

MORE COMPLEX ERROR-HANDLING: `finally`

- Usage of `finally` is used to gracefully crash, examples would be:
  - you opened a file, a problem occurred but it needs to be **closed before stopping**
  - script saved temporary results: **clean up**

INPUT AND OUTPUT

INPUT/OUTPUT WITH PYTHON (I/O)

- Command line arguments
- We can read and writes files to and from the hard disk
  - text files
  - formatted text: xml, html, markdown, postscript
  - multimedia: music, images, video

INPUT/OUTPUT WITH PYTHON (I/O)

- Command line arguments
- We can read and writes files to and from the hard disk
  - text files
  - formatted text: xml, html, markdown, postscript
  - multimedia: music, images, video
- Interactive input events:
  - keyboard and mouse
  - we have to constantly “wait” for them? How?
- Graphical user interfaces

EXECUTING A SCRIPT IN THE TERMINAL OR COMMAND LINE

- Command line arguments can be found with `sys.argv`
- Code to print the arguments:

```python
import sys
for i, arg in enumerate(sys.argv):
  print(f"Argument {i} = {sys.argv[i]}")
```

- Look at the Python script file: `lecture_11_ex_command_line.py`

EXECUTING A SCRIPT IN THE TERMINAL OR COMMAND LINE

- More advanced argument handling using `argparse`
    - define type of argument
    - help description
    - positional vs. named arguments
- Look at the Python script file: `lecture_11_ex_argparse.py`

READING FILES FROM THE HARD DISK

- Make sure the file path exists
- Be careful with file path separators
  - Unix/Linux uses / symbols (forward slash)
  - Windows uses \ symbols (backslash)
  - The \ separator should be escaped \\

```python
# Problem with file paths in Windows:
file_path_in_windows = "C:\Users\mich\Documents\my_file.t
file_path_in_linux = "/home/mich/Documents/my_file.txt"
```

Python expects / or \\ as file separator !

READING FILES FROM THE HARD DISK

- Make sure the file path exists
- Be careful with file path separators
  - Unix/Linux uses / symbols (forward slash)
  - Windows uses \ symbols (backslash)
  - The \ separator should be escaped \\
- While you occupy a file for reading/adapting it might not be accessible by others
  - Important to close the file after finishing
  - What happens if your script crashes while it was writing to a file?

TEXT FILES

- Opening the file and closing after done
- When do you use this method:
    - Multiple separate read/write access
    - for a longer time, doing complex actions

```python
file_path = "samples/sample_text_file.txt"
file = open(file_path)
print(file.read())
file.close()
```

```
This is the first line of the file.
The second line ...
I like banana cake !
```

TEXT FILES

- Opening using the `with` keyword
- Automatically closes the file after it

```python
file_path = "samples/sample_text_file.txt"
with open(file_path) as file:
  print(file.read())
```

```
This is the first line of the file.
The second line ...
I like banana cake !
```

HANDLING FILES THAT DON’T EXIST

- Opening using the **open** keyword
- Ensure that file is closed afterwards by `finally`

```python
file_path = "samples/sample_text3_file.txt"
try:
  file = open(file_path, "r")
  print(file.read())
except FileNotFoundError:
  print("The file does not exist")
except:
  print("Some issue occurred during file reading")
finally:
  file.close()
```

```
The file does not exist
```

VECTOR GRAPHICS: SVG-FILES

- Open the file in PyCharm

```python
file_path = "samples/sample_vector_graphics.svg"
with open(file_path) as file:
  print(file.read())
```

```
<svg version="1.1"
     width="400" height="400"
     xmlns="http://www.w3.org/2000/svg">

  <rect width="100%" height="100%" fill="red" />
  <circle cx="150" cy="100" r="80" fill="green" />
  <text x="150" y="125" font-size="60" text-anchor="middle"
fill="white">SVG</text>

</svg>
```

KEYBOARD AND MOUSE INPUT

- Install the package **pynput**
- Look up the documentation to control keyboard and mouse, example:

```python
from pynput.mouse import Button, Controller

mouse = Controller()

# Read pointer position
print(f"Mouse position is {mouse.position}")

# Set pointer position
mouse.position = (100, 200)
print(f"Mouse moved to {mouse.position}")
```