Osnova

Lesson Overview

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / LESSON OVERVIEW

Bugs, mistakes, errors everywhere.. No more :-)

In this lessons, we will teach you:

- how to **detect various types of errors**,
- ways to **repair incorrect code**,
- how to detect an **error-part of code**.

What about the project from the last lesson? How was it? Let us know.

100% z Lekce 11

Osnova

REVIEW EXERCISES

List Divisors

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / REVIEW EXERCISES / LIST DIVISORS

We would like to find out, what numbers in a given range are divisible by each of the numbers btw. 2 and 9. Your task is to create a program that will:

- collect the information about numbers divisible by each single digit number between 2 and 9
- print the information in nicely formatted table

100% z Lekce 11

Example of program use:

Osnova

```
~/PythonBeginner/Lesson9 $ python list_divisors.py
START POINT: 23
END POINT: 36
| Divisor |       Numbers Divided        |
========================================
|    2    | 24, 26, 28, 30, 32, 34, 36 |
|    3    |     24, 27, 30, 33, 36     |
|    4    |       24, 28, 32, 36       |
|    5    |         25, 30, 35         |
|    6    |         24, 30, 36         |
|    7    |           28, 35           |
|    8    |           24, 32           |
|    9    |           27, 36           |
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Function main()

The workflow of the `main()` function could be:

1. generate a table
2. print the table

```python
def main():
    start,stop = 23,36
    table = make_table(start, stop)
    print_table(table)
```

Osnova

```
    print()
    start = int(input('START POINT: '))
    stop = int(input('END POINT: '))
    print()

    table = make_table(start, stop)
    print_table(table)
```

To create a required table, we will probably want to generate a following list:

```
[[2, '24, 26, 28, 30, 32, 34, 36'],
 [3, '24, 27, 30, 33, 36'],
 [4, '24, 28, 32, 36'],
 [5, '25, 30, 35'],
 [6, '24, 30, 36'],
 [7, '28, 35'],
 [8, '24, 32'],
 [9, '27, 36']]
```

Function make_table()

The first item on each row represent the divisor and the other item is the string of numbers divisible by the divisor. What our `make_table()` function would need to do, to generate such a list could be:

1. Create a variable, where the result will be stored - we call it `table` - we already include the header in it
2. Try to divide each number in the given `range(start,stop)` by numbers 2-9 and collect those that are divisible
3. Append the divisor and divisible numbers to the table

```
def make_table(start, stop,header = ['Divisor','Numbers Divided']):

    rng = range(start,stop+1)
    table = [header]
```

100% z Lekce 11

```
le.append([i,numbers_divisible])

    return table
```

Function get_numbers_divisible()

We have decided to outsource the step number 2 to the function `get_numbers_divisible()` that directly generates a string of divisible numbers by a given divisor:

```
def get_numbers_divisible(divisor, rng):
    numbers = []
    for num in rng:
        if num % divisor == 0:
            numbers.append(str(num))
    return ', '.join(numbers)
```

Function print_table()

Finally, we just implement the printing - formatting functionality:

```
def print_table(table):

    widths = column_widths(table)
    for row in table:
        print('| {:^{w1}} | {:^{w2}} |'.format(*row,**widths))
```

If we wanted to have the table header separated from the body, we could adjust the `print_table()` function as follows - using the `enumerate()` function and the `if i==0` condition:

```
def print_table(table):

    widths = column_widths(table)
    for i,row in enumerate(table):
        print('| {:^{w1}} | {:^{w2}} |'.format(*row,**widths))

```

```
|'.format(*row,**widths))
        print("=" * row_width)
```

Osnova

Function column_widths()

Once again, our `print_table()` function needs information about column widths, that is provided by a helper function:

```
def column_widths(table):
    num_cols = len(table[0])
    widths = {}

    for row in table:
        for col_num,col in enumerate(row):

            key = 'w%d'%(col_num+1)
            widths[key] = widths.setdefault(key,0)
            width = len(str(col))

            if widths[key] < width:
                widths[key] = width

    return widths
```

The whole script

```
def make_table(start, stop,header = ['Divisor','Numbers Divided']):
    rng = range(start,stop+1)
    table = [header]

    for i in range(2,10):
        numbers_divisible = get_numbers_divisible(i, rng)
        table.append([i,numbers_divisible])


    return table
```

100% z Lekce 11

```
def get_numbers_divisible(divisor, rng):
Osnova [illegible] = []
    for num in rng:
        if num % divisor == 0:
            numbers.append(str(num))
    return ', '.join(numbers)


#====================================================

def print_table(table):

    widths = column_widths(table)
    for i,row in enumerate(table):
        print('| {:^{w1}} | {:^{w2}} |'.format(*row,**widths))
        if i==0:
            row_width = len('| {:^{w1}} | {:^{w2}}
|'.format(*row,**widths))
            print("=" * row_width)


def column_widths(table):
    num_cols = len(table[0])
    widths = {}

    for row in table:
        for col_num,col in enumerate(row):

            key = 'w%d'%(col_num+1)
            widths[key] = widths.setdefault(key,0)
            width = len(str(col))

            if widths[key] < width:
                widths[key] = width

    return widths

#====================================================

```

100% z Lekce 11

Osnova

```
    start = int(input('START POINT: '))
    stop = int(input('END POINT: '))
    print()

    table = make_table(start, stop)
    print_table(table)

main()
```

ONSITE EXERCISES

100% z Lekce 11

We will be working with the following three functions:

```
def count(sequence, target=None):
    if not target:
        return len(sequence)
    else:
        count = 0
        for item in sequence:
            count += item==target
    return count


def my_sum(sequence):
    result = 0
    for i in sequence:
        result += i
    return result

def avg(sequence):
    return my_sum(sequence) / count(sequence)
```

What will happen if we run the following commands:

```
>>> avg([1,2,3,4])
```

```
>>> avg([1,2,3,4,'Hello'])
```

Structures to Handle Errors

try - except:

```
try:
    surveilled code
except:
```

Exampl...

Osnova

```
try:
    '2' / 2
except:
    print('There was an error')
```

How can we apply this to our problem?

try - except ExceptionName:

```
try:
    surveilled code
except ExceptionName:
    handled exception
```

Example:

```
try:
    '2' / 2
except TypeError:
    print('There was a TypeError')
```

How can we apply this to our problem?

try - except (Exception1, Exception2):

```
try:
    surveilled code
except (Exception1, Exception2):
    handled exception
```

Example:

```
try:
    num = int(input('Enter a number: '))
    print(num / 2)
except (ValueError, TypeError):
```

100% z Lekce 11

How can we apply this to our problem?

try - except - else:

```
try:
    surveilled code
except ExceptionName:
    handled exception
else:
    code executed if no error raised
```

Example:

```
try:
    num = int(input('Enter a number: '))
    print(num / 2)
except (ValueError, TypeError):
    print('Enter numbers only')
else:
    print('The result is', result)
```

How can we apply this to our problem?

try - finally:

```
try:
    surveilled code
finally:
    executed always
```

Example:

```
def file_write(file,content):
    try:
        file.write(content)
    finally:
        file.close()
```

```
 it'    '''le' e) as file:
        Osnova  e( ntent)
```

try - except - else - finally:

```
try:
    surveilled code
except Exception1:
    handled Exception1
except  Exception2:
    handled Exception2
else:
    additional code if all went well
finally:
    executed always
```

Example:

```
file = open(filename,'w')
try:
    inpt =input('Please enter a number: ')
    num =  int(inpt)
    result = num**2
except TypeError:
    print('Operation not supported for', inpt )
else:
    file.write(result)
finally:
    file.close()
```

How can we apply this to our problem?

RaiseException

We can use exceptions to signal not only bugs, but also a specific situation. To be able to do that, we need to raise errors by ourselves:

100% z Lekce 11

```
Osnova    I do not know you')
```

Debugging

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ONSITE EXERCISES / DEBUGGING

Store the error object

```
except ExceptionName as variable:
    refer to variable
```

This gives us access to error type and message:

```
>>> try:
...     'str'/2
... except Exception as e:
...     err = e
```

```
>>> err.__class__.__name__
'TypeError'
```

```
>>> err.args
("unsupported operand type(s) for /: 'str' and 'int'",)
```

sys.last_traceback

```
>>> avg([1,2,3,'a'])
```

Traceback Attributes:

- `tb_frame` - execution frame object at this level
- `tb_lasti` - index of last attempted instruction in bytecode - for us not so important
- `tb_lineno` - current line number in Python source code

100% z Lekce 11

[illegible] `tb_frame` and `tb_next` but may seem to complicated at the beginning:


```
>>> tb.tb_frame.f_lineno
1
>>> tb.tb_next.tb_frame.f_lineno
2
>>> tb.tb_next.tb_next.tb_frame.f_lineno

4
```

sys.exc_info()

Only inside the except block. We get access to:

- error_type
- error_value
- traceback

```
import sys
>>> try:
...     avg([1,2,3,'a'])
... except:
...     print(*sys.exc_info(), sep='\n')
...
<class 'TypeError'>
unsupported operand type(s) for +=: 'int' and 'str'
<traceback object at 0x7f9bad46eb08>
```

traceback.extract_tb(tb)

Expect input of traceback object. Returns a list of `FrameSummary` objects. Each `FrameSummary` object provides us with information about:

- `filename` - in what file can we find a given function
- `line` - string representing the line of code, where the execution stopped

This page is the most complete source for our custom logging solution.

Osnova

ERRORS

Introduction

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / INTRODUCTION

Meanwhile writing code, programmers make errors:

- logical (mistakes in program design) or
- syntactical (mistakes in code grammar).

Errors cause malfunctioning of programs and therefore programmers want to have them under

100% z Lekce 11

Osnova

... this requirement is raising of so called **exceptions** when an error occurs. ...raised, Python gathers information about it.

At that moment,

- programmer can write code in a way that will handle the error, and the program can continue without interruption
- or the code does not handle the error and outputs the gathered information about the error in a small summary called **traceback**. Tracebacks therefore provide instant feedback to the programmer about bugs in the code.

Exception

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / EXCEPTION

Exception is a **Python representation** of an error. Errors are caused by various causes. Therefore exceptions are categorized based on their cause. Experienced Python programmer knows, in which scenarios a specific type of error can occur and designs the code accordingly.

Errors should **not be always** handled - sometimes it is better that the application crashes. On the other side, from user's point of view it is good to experience as few crashes as possible and thus maybe handle exceptions sometimes.

Exception Types

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / EXCEPTION TYPES

So how the errors actually occur? Let's try to run the following code:

```
>>> result = nonexistent_var**2
```

The code above is trying to retrieve data from nonexistent_var variable, which was not previously

100% z Lekce 11

Therefore the exception `NameError` is raised. This is documented using so called **traceback**:

Osnova

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nonexistent_var' is not defined
```

Exceptions are divided into multiple categories based on the error that precedes them. In the few following slides, we will show those that occur at most at the beginning of programming career.

Type Error

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / TYPE ERROR

If a specific data type does not support a given operation `TypeError` is raised:

Example of expression that could cause such error:

```
>>> '8' / 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

We cannot perform division operation on string data type.

Zero Division Error

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / ZERO DIVISION ERROR

Division by zero is not defined in mathematical world and therefore neither Python knows, how to treat such operation. Intents to divide number by zer invoke `ZeroDivisionError` .

Example:

100% z Lekce 11

```
  F...  ...in...  line 1, in <module>
er...  Osnova  ...r...  : division by zero
```

Syntax Error

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / SYNTAX ERROR

This type of error occurs most often at the beginning, when programmers learn the language or are caused by typos. If we break Python syntax rules in our code `SyntaxError` is raised:

Example:

```
>>> if 5 > 4 print(5)
  File "<stdin>", line 1
    if 5 > 4 print(5)
                 ^
SyntaxError: invalid syntax
```

Conditional statement syntax requires a colon at the end of the header. In our example, the colon is missing.

Syntax errors differ from the rest in that, that they are caught before our code is actually run. So no files are created nor deleted, if that was something our code should have done.

Indentation Error

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / INDENTATION ERROR

In situations when we forget to indent statements properly we get an `IndenationError` :

Example:

```
>>> if 5 ...
```

100% z Lekce 11

```
  F      in    line 2
      Osnova
         ^
IndentationError: expected an indented block
```

In the example above, we should have indented the print statement following the if statement header.

Indentation error belongs into a wider category of errors - Syntax errors.

Lookup Errors

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / LOOKUP ERRORS

There are two kinds of look up errors:

- `IndexError`
- `KeyError`

If we are trying to access item in a sequence using indexing operation, however, the index we provide is higher than the sequence length -1 `IndexError` is raised

Example:

```
>>> [0,1,2,3,4][5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Similarly, if we request a key, that does not exist in a dictionary, we get a `KeyError` :

```
>>> d['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

```
>>      ...e' 'No such key')
No
```


Keyboard Interrupt Error

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERRORS / KEYBOARD INTERRUPT ERROR

Program execution terminated by pressing `Ctrl + c` shortcut is signalled by `KeyboardInterrupt` .

```
>>> while True:
...     print('loop')
...
loop
loop
...
loop
loop
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

To get a more complete overview of what built-in exceptions are there, check [Python Documentation]

Osnova

ERROR HANDLING

Catching Errors

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / CATCHING ERRORS

In Python, exceptions are handled using `try` **statement**. Try statement is another compound statements similarly to `if` statement, `for` loop etc. This means that it contains header(s) and suites.

The most basic try statement looks as follows:

```
try:
    code checked for errors
except:
    code executed if error occured
```

If code inside the `try` block raises an error, the `except` clause catches the error and the code inside the `except` block is executed.

Example:

```
>>> try:
...     inpt = input('Please enter a number: ')
...     num = int(inpt)
... except:
```

100% z Lekce 11

```
Ple        a  mber: abc
      Osnova
 le        n   ers only
```

Operation `int(inpt)` raised `ValueError`, which has been caught by the `except` statement and the code inside it has been executed. We can never trust that users will enter the correct input, therefore it is a good practice to check that part of our program inside `try` statement.

Note that we cannot use the `try` statement without the `except` branch, otherwise we'd get a `SyntaxError`:

```
>>> try:
...     inpt = input('Please enter a number: ')
...     num = int(inpt)
...
  File "<stdin>", line 4
    ^
SyntaxError: invalid syntax
```

Catching Specific Errors

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / CATCHING SPECIFIC ERRORS

The `except` part of the `try-except` statement can be supplemented by the name of a specific exception. Example in the previous section was ready to catch any exception raised inside the its `try` part. This was due to omitting any exception name following the `except` keyword. We could however specify, what error do we want to catch:

```
>>> try:
...     inpt = input('Please enter a number: ')
...     num = int(inpt)
... except ValueError:
...     print('Please enter numbers only')
...
Please enter a number: abc
```

Why t... if ... concrete error name? There are multiple reasons why we should specify the name ... w... would like to handle.

Osnova

Multiple Except Clauses

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / MULTIPLE EXCEPT CLAUSES

The first reason to include error name is that we probably want **different program response to different errors**. Python allows us to specify multiple `except` statements especially for this purpose. The code below checks for both possible incorrect inputs - non-numeric strings and zero for divisor part:

```
>>> try:
...     dividend = int(input('Please enter a dividend number: '))
...     divisor = int(input('Please enter a divisor number: '))
...     result = dividend / divisor
...     print(result)
... except ValueError:
...     print('Please, enter numbers only')
... except ZeroDivisionError:
...     print('Divisor cannot be 0')
...
Please enter a dividend number: abc
Please, enter numbers only
```

Or if we entered correct divident, but incorrect divisor:

```
...
Please enter a dividend number: 3
Please enter a divisor number: 0
Divisor cannot be 0
```

Only the first matching except clause code block is executed. Even though, there could be multiple matches.

100% z Lekce 11

Osnova

...clude error name in except clause is that we maybe want to **catch only** ... **other exceptions to actually terminate our program**. The `try-except` statement is used to handle an error in a program in a graceful way, thus silencing it in a way. However, **especially during debugging period we want to know, what errors occur** in our program. Therefore the general `except` clause is often a bad idea.

Multiple Error Names in One Except

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / MULTIPLE ERROR NAMES IN ONE EXCEPT

Besides having multiple `except` clauses, each catching one type of an error, we can have `except` clauses that apply to multiple error types. `except` clause that would catch multiple error types has to have those errors listed in a tuple:

```
>>> try:
...     dividend = int(input('Please enter a dividend number: '))
...     divisor = int(input('Please enter a divisor number: '))
...     result = dividend / divisor
...     print(result)
... except (ValueError, ZeroDivisionError):
...     print('Please, enter only numbers greater than 0')
...
Please enter a dividend number: abc
Please, enter only numbers greater than 0
```

It depends on concrete scenario, when multiple vs. single error clauses should be applied. All depends on, what actions should be performed after a specific error is raised. If for multiple errors, we need to perform the same action, then it is better to list them inside one except clause.

Exceptions Hierarchy

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / EXCEPTIONS HIERARCHY

100% z Lekce 11

There [illegible] exceptions and some of them pertain to the same family (group). Some [illegible] general and some less. For example `OSError` is a more general exception than `FileNotFoundError`. Thus if we run the below code, we will learn, that our mistake is caught by both `except OSError:` as well as `except FileNotFoundError:` clauses.


```
try:
    file = open('nonexistent.txt')
    print(file.read())
    file.close()
except OSError:
    print('Catching all OSError children')
```

If we ran the above script, we catch the `FileNotFoundError` that belongs into the family of `OSErrors`:

```
Lesson7 $ python catching_child.py
Catching all OSError children
```

Changing the except clause to `except FileNotFoundError` does the same thing - we have just adjusted printed message inside the except clause:

```
try:
    file = open('nonexistent.txt')
    print(file.read())
    file.close()
except FileNotFoundError:
    print('Could not find the file')
```

```
Lesson7 $ python catching_child.py
Could not find the file
```

To understand the above observation, we need to learn, that **errors (objects) are in Python system organized in a hierarchy**. There is one exception type at the top of the hierarchy called `BaseException`. If applied to `except` statement, each exception would be caught. That means that more general error types in `except` clause apply to all the exception types, that fall in the hierarchy below them. Actually, leaving `except` clause without any error name is the same as putting there `BaseException`.

```
...            nonexistent * 5
...    Osnova  Exception:
...     print('Catching anything')
...
Catching anything
```

The above is the same as:

```
>>> try:
...     a = nonexistent * 5
... except:
...     print('Catching anything')
...
Catching anything
```

The most general exception clause we should use in our programs is `except Exception` .

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
```

To learn the whole exception hierarchy see Python Documentation

Finally Clause

Another clause available for handling exceptions is the `finally` statement. The most basic syntax required to use finally clause is a combination of `try-finally` statement:

```
try:
   statements
finally:
```

Another [illegible] syntax is the combination of `try-except-finally` :


```
try:
   statements
except:
    statements
finally:
   statements
```

`finally` clause is executed always, no matter whether an exception has been raised inside the `try` block or not.

Finally Clause Example

The `try` block below does two things:

- open a file based on user input
- implicitly tries to convert the content of the file into a dictionary

```
try:
    filename = input('Please enter the name of the file you want to open:
')
    file = open(filename)
    db = eval(file.read())
    yearly_salary= db['XC4332']['salary'] * 12
except FileNotFoundError:
    print('Could not find the file')
finally:
    print('Closing the file')
    file.close()
```

There are two possible outcomes:

2. File was opened but the key could not be find - `KeyError` raised

Osnova

3. File was opened, but during the operation performed on its content, `TypeError` occured.

In the last two cases it would be great to have place, where the file would be explicitly closed before leaving the program.

If we ran the file our program ends up in error, but the file has been correctly closed - see the `Closing the file` line:

```
Lesson7 $ python catching_child.py
Please enter the name of the file you want to open HRS/employees.txt
Closing the file
Traceback (most recent call last):
  File "catching_child.py", line 5, in <module>
    yearly_salary= db['XC4332']['salary'] * 12
KeyError: 'XC4332'
```

Resources as files are usually closed inside the `finally` block. It is because of the fact that `finally` block is executed always, even though our program would have crashed. Thus connections do databases, files etc. can be safely managed.

Else Clause

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / ELSE CLAUSE

The error handling statement can contain, similarly to conditional or loop statements, **else** clause.

Code inside the else statement is executed only if no error was raised while the try block ran. The **else** part has to be **situated after the except statements** as depicted below. Otherwise SyntaxError would be raised:

```
try:
    statements
```

100% z Lekce 11

```
ls
    Osnova
    .i    raised -> statements executed
```

En example if all went well inside the `try` block:

```
>>> try:
...     num1 = int(input('Divident: '))
...     num2 = int(input('Divisor: '))
...     result = num1 / num2
... except (ValueError, ZeroDivisionError):
...     print('Please enter only numbers and non-zero divisors')
... else:
...     print('The result of {}/{} is: {}'.format(num1,num2,result))
...
Divident: 34
Divisor: 2
The result of 34/2 is: 17.0
```

Note that we cannot include the `else` branch unless we also use `except`, otherwise we'd get a `SyntaxError`.

Another Else Example

In the example below, we are trying to open a file. We are handling the possible case of non-existent file by the except clause that informs the user. Otherwise the data is read in from the file, that is, the exception has not been raised and the else clause can be executed.

```
import csv
try:
    fl = open('nonexistent.csv')

except IOError:
```

```
    = `v.reader(fl).read()
Osnova
```

```
Could not find the file
```

Exception Handling Summary

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / EXCEPTION HANDLING SUMMARY

Our most exception catching statement could combine all the types of clauses:

```
try:
    code checked for errors
except Error1:
    statements executed if Error1 raised
except (Error2,Error3):
    statements executed if Error2 or Error3 raised
else:
    statements executed if no error raised
finally:
    statements executed in any case
```

A. If an **exception occurs** inside the try block, Python looks for an except clause that matches the raised exception:

1. **the exception matches** one that the except statement names

   a. statements inside the `except` block run
   b. `finally` clause block, if present, is executed
   c. then program execution resumes below the entire `try` statement

2. **the exception does not match** one that the one of the except statement names

   a. `finally` clause block, if present, is executed
   b. Python kills the program and prints a default error message.

100% z Lekce 11

1. Python runs the statements inside the `else` clause (if present).

2. Python runs the statements inside the `finally` clause (if present)
3. After that the control resumes below the entire `try` statement.

Next, we will go through some **additional info** concerning exceptions.

ERROR HANDLING +

Raising Exceptions

Example:

```
>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError
```

As you may have noticed, there is no message following the exception name in the traceback. We can supply it inside the parentheses appended after the exception name within the raise statement:

```
>>> raise NameError('I do not know you')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: I do not know you
```

We will discuss two use cases, when `raise` statement can be used:

1. We need to raise (activate) programmer defined exceptions
2. We want to re-raise an exception after catching it - in scenarios of chained exceptions

Create own Exception types

We can create a new exception as follows:

```python
class ExceptionName(Exception):
    pass
```

The `class` keyword and the `Exception` inside parentheses are required. We decide, how the exception will be called.

```
 c   ctMove(Exception):
```


In the code it could look like this:

```
def move_piece(piece,destination, board):

    moves_available = generate_moves(piece,board)

    if destination not in moves_available:
        raise IncorrectMove('{} cannot move to {}'.format(piece, destination)

    board[destination] = piece

    return board
```

We had to use `raise` statement to throw the error.

Handling Exceptions from lower scopes

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING + / HANDLING EXCEPTIONS FROM LOWER SCOPES

We probably havent stressed enough the fact, that if the error is not handled in the scope of the try statement, where the error occured, Python propagates it up the call stack until the matching except clause is found or the program is killed.

An example could be a situation, where the `avg()` function needs to check, whether both `my_sum()` and `count()` functions provide correct inputs. Any error that has been thrown inside one of those two functions, will be propagated to the `avg()` execution frame into the `try` block, where it will be handled by one of the `except` clauses.

```
def count(sequence, target=None):
    if not target:
```

Osnova

```
        count = 0
        for item in sequence:
            count += item==target
        return count


def my_sum(sequence):
    result = 0
    for i in sequence:
            result += i
    return result

def avg(sequence):
    try:
        return my_sum(sequence) / count(sequence)
    except ZeroDivisionError:
        print('Zero Division Error has been handled')
    except TypeError:
        print('Type Error has been raised')
```

```
>>> avg([])
Zero Division Error has been handled
>>> avg([1,2,3,'a'])
Type Error has been raised
```

Raising Exception inside Except

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING + / RAISING EXCEPTION INSIDE EXCEPT

It may happen, that during the handling of an exception inside `except` block, another error is raised. Then the error summary report looks as follows:

```
>>> try:
        'str' / 2
```

Osnova

```
...ecent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ZeroDivisionError: division by zero
```

If we have in our error report phrase: `During handling of the above exception, another exception occurred:` , it means that the except block contains problematic code and we should probably fix it.

DEBUGGING

100% z Lekce 11

Osnova ... tro

PYTHON ACADEMY / **9. ERRORS & DEBUGGING** / **DEBUGGING** / **DEBUGGING INTRO**

Debugging is an activity that should lead to **removal of mistakes in our code**. If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of debugging.

Python offers a more sophisticated tool, to record, what activities are performed during the program run. One of them is the `logging` module, another one could be the `pdb` (python debugger) module.

Very basic but important tool that helps programmers to discover errors in their programs is the **traceback**.

Execution Frame

PYTHON ACADEMY / **9. ERRORS & DEBUGGING** / **DEBUGGING** / **EXECUTION FRAME**

A Python program is constructed from code blocks. A block is a piece of Python program text that is executed as a unit. Specifically, we have for example the following code blocks:

- python module (python file)
- a function body

A code block is executed in an **execution frame**. A frame contains information used often for debugging and determines where and how execution continues after the code block's execution has completed. Execution frame is thus a representation of function call or a running Python program.

We can imagine execution frame as a **box**, that keeps track of:

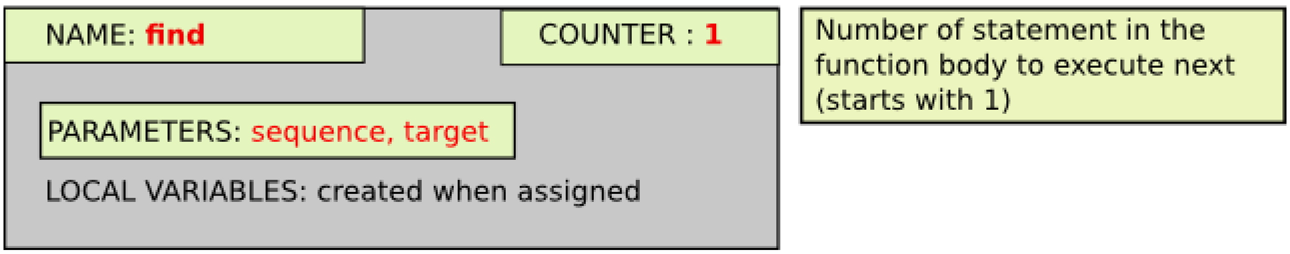
- function name,
- execution counter - what line is being currently executed,

100% z Lekce 11

All this information is then used inside the traceback. An abstraction of execution frame is called

Example of a function and its execution frame created when called:

```
def find(sequence, target):
 for index,item in enumerate(sequence):
  if item == target:
   return index
 return -1
```



There are many other Python concepts, that when run, execution frame is created. We will however not mention them here.

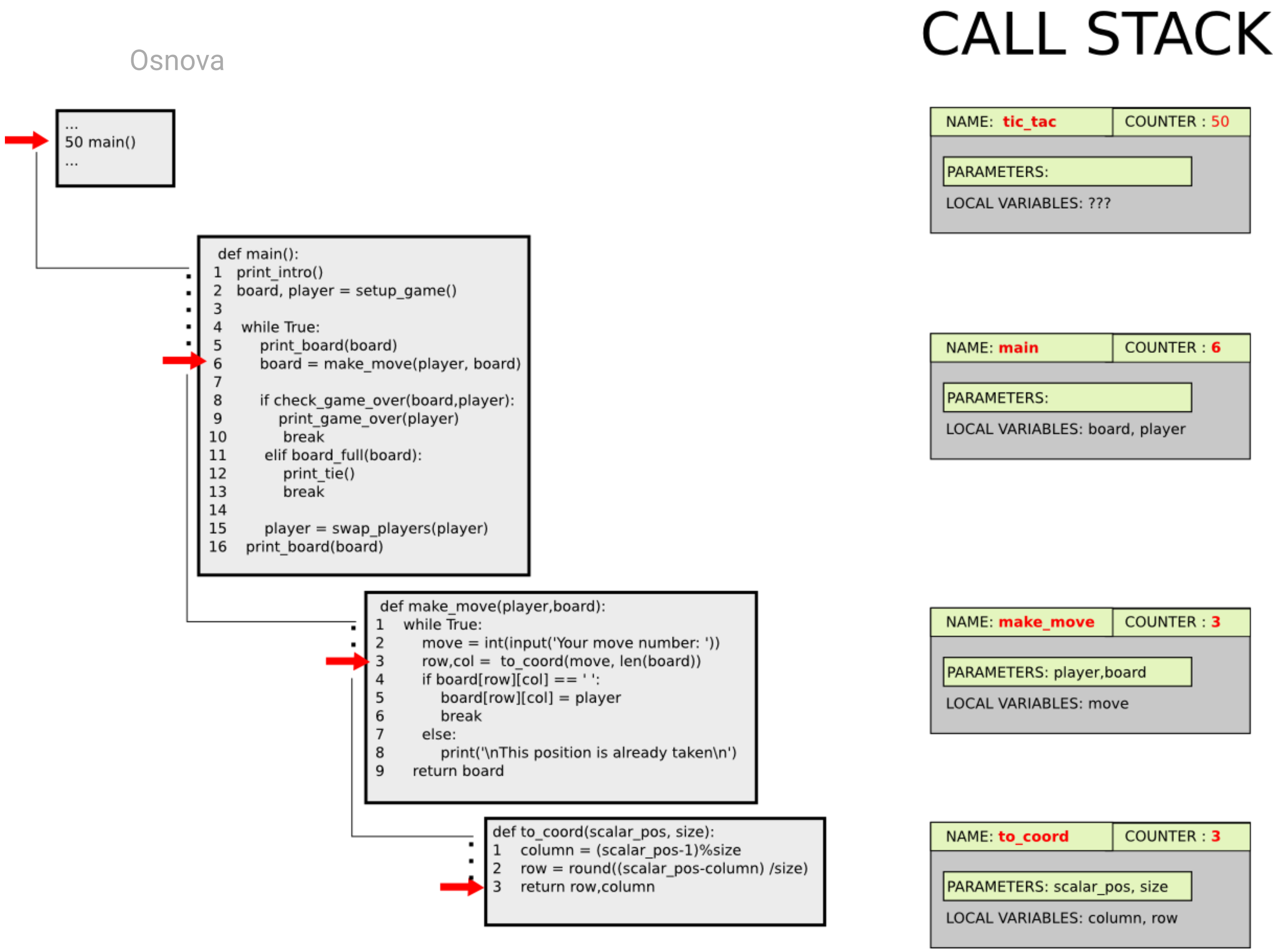
Call Stack

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING / CALL STACK

When a function is called within a function, new execution frame is created. Therefore our tracebacks can list multiple execution frames. This accumulation of execution frames is called **call stack**. We can imagine it as a pile of cards, where on each card is written its name, line, where the call to another function has been executed.

Therefore once the function `to_coord()` is done with its work, its execution frame is destroyed and the program execution continues in the function `make_move()` from line, where the call to `to_coord()` was executed. Once `make_move()` returns, the exection continues in function `main()` and once this one is done, then the module called `tic_tac.py` is finished.

Osnova



Error Report

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING / ERROR REPORT

Error summary is the main resource of information regarding errors in Python. Let's say, we run the following code from a Python file `load_db.py` :

```
file = open('nonexistent.txt')
db = eval(file.read())
yearly_salary = db['XC4332']['salary'] * 12
```

We would get the following error report:

100% z Lekce 11

```
  File "load_db.py", line 1, in <module>
    file = open('nonexistent.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent.txt'
```

Osnova

The report contains the following information:

1. Traceback - summary listing from the top of call stack to the point where the error occured. It helps to identify the file and exact code line, where the error has occured. In our case error occurred on the first line of the file `load_db.py` :

```
Traceback (most recent call last):
  File "load_db.py", line 1, in <module>
    file = open('nonexistent.txt')
```

2. Error Type:

```
FileNotFoundError: ...
```

3. Error Value:

```
...: [Errno 2] No such file or directory: 'nonexistent.txt'
```

Multiple Frames in a Traceback

We have defined 3 functions:

1. `count` - counts, how many times a target encounters itself in a sequence or how long the sequence is
2. `my_sum` - sums all the numbers in a sequence
3. `avg` - calculates the mean of a given sequence

```
def count(sequence, target=None):
```

100% z Lekce 11

Osnova

```
        count = 0
        for item in sequence:
            count += item==target
        return count


def my_sum(sequence):
    result = 0
    for i in sequence:
        result += i
    return result

def avg(sequence):
    return my_sum(sequence) / count(sequence)
```

Function `avg` uses another two functions to calculate the result. When the function `avg` is called, its execution frame is created and put on the call stack. Then the function `my_sum` is called and its execution frame is put on the call stack. Everything works well and both functions return correctly if correct input is provided.

```
>>> avg([1,2,3,4,5])
3.0
```

However, if the sequence contains non-numeric item, we get a `TypeError` that is raised inside the `my_sum` function. The traceback, that is printed to the terminal then contains the listing of the whole call stack from the place, where the error has occured up to the place in main module, where the first call of this call chain has been executed:

```
>>> avg([1,2,3,4,'a'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
  File "<stdin>", line 4, in my_sum
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

We can see, that the traceback lists all the execution frames here and we can trace the cause of

100% z Lekce 11

Osnova

ug

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING / SOLVING THE BUG

Once we know that the fourth line in the `my_sum()` function is vulnerable to errors, we can fix this using `try-except` statement or `if` statement. In the example below, we decided to use `if isinstance(i, int, float):` to check, whether the sequence item belongs to one if the numeric data types :

```
def count(sequence, target=None):
    if not target:
        return len(sequence)
    else:
        count = 0
        for item in sequence:
            count += item==target
        return count


def my_sum(sequence):
    result = 0
    for i in sequence:
        if isinstance(i, (int, float)):
            result += i
    return result

def avg(sequence):
    return my_sum(sequence) / count(sequence)
```

When running the `avg()` function, we get no error and the program behaves as if nothing has happened:

```
>>> avg([1,2,3,4,'a'])
2.0
```

100% z Lekce 11

Besides using `print()` function to signal variable values, we can also use `assert` statement that raises `AssertionError` if a given condition is not `True` . The condition should probably test, whether a given variable's value meets some requirements.

For example, the sequence passed into the `avg()` function should not be empty, otherwise we would be trying to perform division by zero:

```
def avg(sequence):
    assert len(sequence) > 0
    return my_sum(sequence) / count(sequence)
```

When executed, we get the following result

```
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
AssertionError
```

We could also pass a message along with the assertion error, if separate the assertion test from the message by comma:

```
def avg(sequence):
    assert len(sequence) > 0, 'Cannot accept empty sequences'
    return my_sum(sequence) / count(sequence)
```

```
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
AssertionError: Cannot accept empty sequences
```

Use Case

' ar' ' ' 're od for catching false assumptions that were made while writing the code, nc Osnova c ur code by another programmer. In addition, they can act as in-line documentation to some extent, by making the programmer's assumptions obvious.

Assertions can be disabled by passing the -O option when invoking Python script:

```
~/Scripts $ python -O file.py
```

However, assert statements are maybe not used as often as debugger tools and `try-except` statements

DEBUGGING +

Accessing Exception Information

100% z Lekce 11

W... c... ...for...ation about the last error in the following ways:
Osnova

1. `sys.last_traceback`
2. `sys.exc_info`
3. `except ErrorName as e`

We will discuss each of the above options in the following sections. But what information can we get from the error object?

Usually we want to access:

1. traceback
2. information about error type
3. error value - in other words the message, following the error name in the summary report

Why would we want to access information about the error? Mainly for our own reporting purposes. We may want to keep a record of errors that have occured in our application in a text file and we may want to format them in a nice manner.

Storing Exception Object

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / STORING EXCEPTION OBJECT

In case we want to work further with the exception inside the except block, we can assign the error object to a variable. This is done inside the `except` statement using the keyword `as` .

An example:

```
>>> try:
...     'str'/2
... except Exception as e:
...     err = e
...
```

100% z Lekce 11

To access the information about the error, we can do the following:

Osnova

1. **Error type** - use `err.__class__` attribute:

```
>>> err.__class__
<class 'TypeError'>
```

Or to get a name of the error:

```
>>> err.__class__.__name__
'TypeError'
```

2. **Error value** (message) - use `err.args`

```
>>> err.args
("unsupported operand type(s) for /: 'str' and 'int'",)
```

3. **Traceback** - traceback is not stored inside the error object, therefore we cannot retrieve it

sys.last_traceback

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / SYS.LAST_TRACEBACK

After we import module `sys` , we can access information about the last traceback through the variable `last_traceback` . Let's create an error using our avg function:

```
def count(sequence, target=None):
    if not target:
        return len(sequence)
    else:
        count = 0
        for item in sequence:
            count += item==target
        return count

```

100% z Lekce 11

```
        lt    0
        sequence:
            result += i
    return result

def avg(sequence):
    return my_sum(sequence) / count(sequence)
```

Result of calling `avg()` with sequence containing non-numeric items:

```
>>> avg([1,2,3,'a'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
  File "<stdin>", line 4, in my_sum
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Now we can access the above traceback through `sys.last_traceback`. As with error object, we could access only information related to error object, here we can access only traceback information, not the error type and value. The traceback information we can access are:

1. `tb_frame` - execution frame object at this level

```
>>> sys.last_traceback.tb_frame
<frame object at 0x7f9bad46f1e8>
```

2. `tb_lasti` - index of last attempted instruction in bytecode - for us not so important

```
>>> sys.last_traceback.tb_lasti
18
```

3. `tb_lineno` - current line number in Python source code - in our case, running the this command in interactive console returns number 1 as that is our current line number.

```
>>> sys.last_traceback.tb_lineno
1
```

4. `tb_next` - next inner traceback object (called by this level)

If we ... using `tb_next` , we get all the lines from the traceback until we arrive at ... 4 ...


```
>>> sys.last_traceback.tb_next.tb_lineno
2
>>> sys.last_traceback.tb_next.tb_next.tb_lineno
4
```

All the traceback objects that we would retrieve can be easier processed using `traceback` module.

sys.exc_info()

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / SYS.EXC_INFO()

Module `sys` provides a function `sys.exc_info()` , that allows us to access all the information about the error as well as traceback in a tuple: `(error_type, error_value, traceback)` .

This function can be however called **only inside the `except` block**. There exception still exists. Once the execution leaves the `except` body, the exception goes away. Calling the `exc_info()` function outside the `except` block returns a tuple of three `None` values:

```
>>> sys.exc_info()
(None, None, None)
```

As example we will use our `avg()` function from the section on `sys.last_traceback` :

```
import sys
>>> try:
...     avg([1,2,3,'a'])
... except:
...     print(*sys.exc_info(), sep='\n')
...
<class 'TypeError'>
unsupported operand type(s) for +=: 'int' and 'str'
```

... tuple returned by the function `exc_info()` into the `print()` function ... parameter to newline character. We can see that we have access to all the information that would be otherwise printed to the error report.

Osnova

Traceback module

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / TRACEBACK MODULE

Python library contains a module called `traceback` which allows us to:

1. extract, format and print tracebacks in a custom manner
2. extract, format and print error reports according to our needs

We will not discuss all the classes and functions this module offers. You are welcome to consult [Python documentation]() dedicated to this module. In this section we will just demonstrate, how we can access and format information that is present inside the error report summary.

1. We will first create some error using `avg()` function:

```
>>> avg([1,2,3,'a'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
  File "<stdin>", line 4, in my_sum
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

2. Then we will extract traceback from the variable, that contains the information about it using `traceback` and `sys` module:

```
>>> import traceback
>>> import sys
>>> f_summary = traceback.extract_tb(sys.last_traceback)
>>> f_summary
[<FrameSummary file <stdin>, line 1 in <module>>, <FrameSummary file
<stdin>, line 2 in avg>, <FrameSummary file <stdin>, line 4 in my_sum>]
```

100% z Lekce 11

ar ob t reflect the information about a single execution frame. From the above
Osnova
or ca. ce, there are three execution frames in the traceback. Therefore we got back the list of three `FrameSummary` objects. Each `FrameSummary` object provides us with information about:

1. `filename` - in what file can we find a given function
2. `line` - string representing the line of code, where the execution stopped
3. `lineno` - line number, where the error occured
4. `name` - name of the function - execution frame - where the error occured

And here is how does the information looks like:

```
>>> summary[0].name
'<module>'
>>> summary[1].name
'avg'
>>> summary[2].name
'my_sum'
>>> summary[0].filename
'<stdin>'
```

As we are running this commands in interactive console, information about filename is available only to the top most frame and neither the line attribute provides what it should. We will need to demonstrate the above on a Python file.

In the next section, we will put together all the information we have learned so far to make a small logging program.

Our own logging program

Information about error can be gathered only inside the except block, therefore we will use

100% z Lekce 11

...xtract all this information is placed inside the `except` clause, inside


```python
import traceback,sys

def count(sequence, target=None):
    if not target:
        return len(sequence)
    else:
        count = 0
        for item in sequence:
            count += item==target
    return count


def my_sum(sequence):
    result = 0
    for i in sequence:
        result += i
    return result

def avg(sequence):
    try:
        return my_sum(sequence) / count(sequence)
    except Exception:
        f_summaries = traceback.extract_tb(sys.exc_info()[2])

        for s in f_summaries:

            print('FILE {} / FUNCTION: {} / LINE {}: \n\t{}'
                  .format(s.filename,s.name,s.lineno,s.line))
        print('{.__name__}: {}'.format(*sys.exc_info()[:2]))


avg([1,2,3,'a'])
```

```
~/PythonBeginner/Lesson7 $ python test_traceback.py
FILE test_traceback.py / FUNCTION: avg / LINE 24:
    return my_sum(sequence) / count(sequence)
FILE test_traceback.py / FUNCTION: my_sum / LINE 16:
    result += i
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```


Note the `print('{.__name__}: {}'.format(*sys.exc_info()[:2]))` line, where we use attribute `.__name__` inside the curly braces. What is happening here, is that Python iserts the first item from `*sys.exc_info()` variable, which is the error object, and requests the content of its `__name__` attribute. If the error object is `NameError` for example, the code inside curly brackets is doing the following : `NameError.__name__` .

We could stream line our code even further and extract the logging functionality into a logging function `logger()` :

```
def avg(sequence):
    try:
        return my_sum(sequence) / count(sequence)

    except Exception:
        logger(sys.exc_info())

def logger(exception_info):
    summary = traceback.extract_tb(exception_info[2])
    for s in summary:

        print('\nFILE {} / FUNCTION: {} / LINE {}: \n\t{}\n'
                .format(s.filename,s.name,s.lineno,s.line))

    print('{.__name__}: {}'.format(*exception_info[:2]))

avg([1,2,3,'a'])
```

We have added newline character before and after each line in traceback and this is the result:

```
~/PythonBeginner/Lesson7 $ python test_traceback.py
```

```
FILE: ...ack.py / FUNCTION: my_sum / LINE 16:
...
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```


Stream Redirection

Our `logger()` function is for now able to print the output only to the terminal. What if we wanted to have all the information stored inside a file instead of just printing it to the terminal?

We could use so called **stream redirection**. We redirect the stream of written information from one file to another.

What function `print()` is actually doing, when it prints the information to the console is that it writes information to a file-like object called **standard out**.

If we looked at the standard documentatin concerning `print()` function, we would find out, that one of its parameters is called `file` and has default value `sys.stdout`. We could therefore assign this parameter our file object, and all the information will be written there instead of the terminal:

```
def avg(sequence):
    try:
        return my_sum(sequence) / count(sequence)
    except Exception:
        with open('log.txt', 'a') as f:
            logger(sys.exc_info(),file=f)


def logger(exception_info,*, file=sys.stdout):

    summary = traceback.extract_tb(exception_info[2])
    for s in summary:
```

```
                  .format(s.filename,s.name,s.lineno,s.line),
    )

    print('{.__name__}: {}'.format(*sys.exc_info()[:2]),file=file)
```

Osnova

We have used **context manager** inside the `except` clause in the `avg()` function. This will assure that the file will be closed even if an error occured inside `logger()` function. The file object has been opened in **append** mode in order to avoid overwriting of the information.

The `logger()` function has received one mandatory keyword-only parameter `file`, that has default value pointing to the terminal. All the `print()` statements inside this function then receive the value of `file` parameter.

After running our script twice, this is how the file `log.txt` looks like:

```
FILE test_traceback.py / FUNCTION: avg / LINE 50:
    return my_sum(sequence) / count(sequence)


FILE test_traceback.py / FUNCTION: my_sum / LINE 16:
    result += i

TypeError: unsupported operand type(s) for +=: 'int' and 'str'

FILE test_traceback.py / FUNCTION: avg / LINE 50:
    return my_sum(sequence) / count(sequence)


FILE test_traceback.py / FUNCTION: my_sum / LINE 16:
    result += i

TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Additional modules

100% z Lekce 11

Osnova

A[illegible] at the beginning of this section, Python has its additional modules that u[illegible]g & debugging:

- pdb module for debugging which you can check in Python documentation
- loggin module providing logging utilities which you can check in Python documentation or in a Logging HOW TO tutotrial.

QUIZ

Error Handling & Debugging

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / QUIZ / ERROR HANDLING & DEBUGGING

100% z Lekce 11

What is exception?

Osnova

A. Part of the code, that is ignored

B. Exception is a term for error inside a Python program

C. Part of the code, that is executed always, even before the program shut down

D. It is a special object, where we can temporarily store result returned by a function.

HOME EXERCISES

100% z Lekce 11

Osnova

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / HOME EXERCISES / FILE LOADER

Create a function, that will accept one input - file path - and will return the content of the file. If the file path is incorrect - the file does not exist, the function should return value `None` and print message:

```
File <filename> does not exist.
```

In place of `<filename>` should be inserted the actual name of the file - not the whole path. The file path can be absolute or relative.

The file should be closed inside the `file_loader()` function before returning. Use exception handling statements to solve this task.

Example of using the function:

```
>>> c = file_loader('non')
File non does not exist.
>>> c = file_loader('test.txt')
>>> c
''
>>> c = file_loader('log.txt')
>>> print(c)
This is the first line
```

Python Online Editor

```
1
```

100% z Lekce 11

Osnova

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Our solution could be to try to open the file inside the `try` block. If the `open()` function raises `FileNotFoundError` we catch it using the `except` statement. At that moment no file is opened so we just extract the file name from the path and print the message.

If no error has been raised during opening of the file, we read its content, close it and return the content.

```
def file_loader(filepath):

    try:
        file = open(filepath)

```

100% z Lekce 11

```
            print('File {} does not exist'.format(filename))
        else:
            content = file.read()
            file.close()
            return content
```


Roman & Arab Numerals

Create 2 functions:

1. function `to_roman()` that will convert any roman numeral into its arabic equivalent
2. function `to_arab()` that will convert any arab number into its roman numeral equivalent

Roman digits	Arabic Equivalent
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

Make sure also, to keep the rule of not repeating for times `'I'` or `'X'` in numbers like 4, 9, 40, 90, etc.:

Roman digits	Arabic Equivalent
IV	4

Osnova R...n digits	Arabic Equivalent
XL	40
XC	90
CD	400
CM	900

Few rules

- numeral I is placed before V or X indicates one less, so four is IV (one less than five) and nine is IX (one less than ten)
- numeral X placed before L or C indicates ten less, so forty is XL (ten less than fifty) and ninety is XC (ten less than a hundred)
- numeral C placed before D or M indicates a hundred less, so four hundred is CD (a hundred less than five hundred) and nine hundred is CM (a hundred less than a thousand)

And what about numbers greater than 1000? Our converters should work up to number 3999 what in roman numerals looks like MMMCMXCIX.

Example of functions in use:

```
>>> to_arab('MMMCMXCIX')
3999
>>> to_roman(3999)
'MMMCMXCIX'
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

100% z Lekce 11

Osnova

Function to_arab()

The key to the solution of this task may be to realize, we can use a dictionary to map arab numerals to roman numerals.

To convert roman numeral to arab, we just need to sum all the numbers from left to right. We should take care of cases, when there are numerals consisting of two letters, e.g. `'IV'` , `'IX'` etc., but represented by one arabic digit. For those we may prefer to create their own mapping.

In the loop, where we will sum all the numbers, we will just check, whether the next two letters in roman numerals can be represented by one arab digit. Based on this check we will extract two or one letter from the beginning of the roman numeral.

```
def to_arab(roman):
    mapping1 = {'IV':4, 'IX': 9,'XL':40, 'XC':90,
                'CD':400, 'CM':900}
    mapping2 = {'I':1, 'V':5, 'X': 10, 'L': 50,
                'C':100, 'D':500, 'M':1000}
    result = 0

    while roman:
        if roman[:2] in mapping1:
            num = roman[:2]; roman=roman[2:]
            mapping = mapping1
        else:
            num = roman[:1]; roman = roman[1:]
            mapping = mapping2

        result += mapping[num]

    return result
```

Function to_roman()

100% z Lekce 11

Osnova

1. Take the highest value among the keys in the mapping and store it to variable `arab`
2. Variable `arab` provides the key to the roman numeral, which we will store into the variable `roman`
3. If the input number `num` would be 3999, for example, the first extracted arab digit would be 1000 and roman numeral 'M'. To find out, how many times number 1000 encounters itself in 3999, we use floor division operation -> `num//arab` -> 3999 // 1000 -> what is equal to 3. We therefore know, that letter 'M' should be in our newly built numeral three times. Therefore we add it three times to the `result` string (`result += num // arab * roman`)
4. As we have already used 3000 out of 3999, we need to keep only the remainder of the original number in the variable `num` . Therefore we use the operation `num %= arab` , what is 3999 % 1000 => 999.

We gradually build the roman numeral performing the above four steps.

```python
def to_roman(num):
    result = ''
    mapping = { 1:'I',4:'IV',5:'V',9:'IX',10:'X',
                40:'XL', 50:'L',90:'XC',100:'C',
                400:'CD', 500:'D',900:'CM',1000:'M'}

    for arab in sorted(mapping, reverse=True):

        roman = mapping[arab]
        result += num // arab * roman
        num %= arab

    return result
```

Horizontal Histogram

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / HOME EXERCISES / HORIZONTAL HISTOGRAM

100% z Lekce 11

l· ·s · · f · · ch i· ·m. Each line should contain as many stars * as is the magnitude of the
· ·nu · · · Osnova 4 stars). Thus we will create columns.

The histogram should have Y axis on its left border.

Example of function in use:

```
>>> horizont_hist([4,5,7,10,6,3,2])
|****
|*****
|*******
|**********
|******
|***
|**
```

Online Python Editor

```
1
```

100% z Lekce 11

Osnova

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

```
def horizont_hist(data):

    for value in data:

        print('|' + '*' * value)
```

Vertical Histogram

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / HOME EXERCISES / VERTICAL HISTOGRAM

Create a program that will print historgram as the one below. The function that generates the histogram should take one input - list of values. The histogram should then depict the magnitudes of those values.

If a histogram receives the following list `[4,5,7,10,6,3,2]`, then it should depict those magnitudes as follows:

```
~/PythonBeginner/Lesson7 $ python charts.py
 10|              **
  9|              **
```

100% z Lekce 11

```
  7|              **
  6|              **  **
  5|      **  **  **  **
  4|  **  **  **  **  **
  3|  **  **  **  **  **  **
  2|  **  **  **  **  **  **  **
  1|  **  **  **  **  **  **  **
```

Bonus

Function that generates the histogram may receive an input telling it, how wide the columns should be.

Example:

```
~/PythonBeginner/Lesson7 $ python charts.py
 10|                 ***
  9|                 ***
  8|                 ***
  7|            ***  ***
  6|            ***  ***  ***
  5|       ***  ***  ***  ***
  4|  ***  ***  ***  ***  ***
  3|  ***  ***  ***  ***  ***  ***
  2|  ***  ***  ***  ***  ***  ***  ***
  1|  ***  ***  ***  ***  ***  ***  ***

~/PythonBeginner/Lesson7 $ python charts.py
 10|                    ****
  9|                    ****
  8|                    ****
  7|              ****  ****
  6|              ****  ****  ****
  5|        ****  ****  ****  ****
  4|  ****  ****  ****  ****  ****
```

1' **** **** **** **** **** ****
Osnova

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

To solve any task, it is good to think about the output that the function should provide. We have decided, that our `vert_histogram()` function should return a list of lists, that would look like the one below:

```
[[' 10|', '  ', '  ', '  ', '**', '  ', '  ', '  '],
 ['  9|', '  ', '  ', '  ', '**', '  ', '  ', '  '],
 ['  8|', '  ', '  ', '  ', '**', '  ', '  ', '  '],
 ['  7|', '  ', '  ', '**', '**', '  ', '  ', '  '],
 ['  6|', '  ', '  ', '**', '**', '**', '  ', '  '],
 ['  5|', '  ', '**', '**', '**', '**', '  ', '  '],
 ['  4|', '**', '**', '**', '**', '**', '  ', '  '],
 ['  3|', '**', '**', '**', '**', '**', '**', '  '],
 ['  2|', '**', '**', '**', '**', '**', '**', '**'],
 ['  1|', '**', '**', '**', '**', '**', '**', '**'],
 ['    ________________________________']]
```

All we would need to do to print it nicely to the terminal is to repeatedly use `str.join()` method on each row and print the result of that operation.

The last row of our histogram list is a X axis.

So how can we get to the above result?

Function vert_histogram()

This function should probably do the following:

100% z Lekce 11

the ...nerated row (one per iteration) to the "mother" list stored under the

Osnova

```
def vert_histogram(data,col_width=2):


    LABEL_WIDTH = len(str(max(data))) + 1
    DATASET_SIZE = len(data)

    hist = []
    current_row_num = max(data)

    while current_row_num > 0:

        row = generate_row(current_row_num,col_width, data,
                           LABEL_WIDTH, DATASET_SIZE)
        hist.append(row)

        current_row_num -= 1

    hist.append(make_baseline(col_width,LABEL_WIDTH,DATASET_SIZE))

    return hist
```

Function generate_row()

Row consists of a label - number value to the left of the Y axis and individual columns. Our function below

1. fist creates the label item
2. creates a list, that contains of the label and number of places, where part of the column may be drawn

```
['  4|', '  ', '  ', '  ', '  ', '  ', '  ', '  ']
```

After generating the whole level for the value 4, the result should be as follows:

An' '' '''ing f''ction could look like this:
Osnova

```
def generate_row(current_row_num,col_width, data,LABEL_WIDTH,
DATASET_SIZE):

    label = '{: >{w}}|'.format(current_row_num,w=LABEL_WIDTH)
    row = [label] + [' '*col_width] * DATASET_SIZE

    for i in range(DATASET_SIZE):
        if data[i] >= current_row_num:

            row[i+1] = '*' * col_width

    return row
```

Function make_baseline()

This function just generate X axis for our chart.

```
def make_baseline(col_width,DATASET_SIZE):

    label_offset = [' ']* col_width

    line = ['_'*(col_width+2)]*DATASET_SIZE

    return [''.join(label_offset +line)]
```

Function print_histogram()

Each line is printed separately to the terminal using the `for` loop.

```
def print_histogram(hist):
    print()
    for line in hist:
        print('  '.join(line))
```

100% z Lekce 11

At [illegible] cal[illegible] the `vert_histogram()` and `print_histogram()` functions:

Osnova

```
hist = vert_histogram([4,5,7,9,6,3,2],4)
print_histogram(hist)
```

```
~/PythonBeginner/Lesson7 $ python charts.py
 10|              **
  9|              **
  8|              **
  7|          **  **
  6|          **  **  **
  5|      **  **  **  **
  4|  **  **  **  **  **
  3|  **  **  **  **  **  **
  2|  **  **  **  **  **  **  **
  1|  **  **  **  **  **  **  **

   ________________________________
~/PythonBeginner/Lesson7 $ python charts.py
 10|                 ***
  9|                 ***
  8|                 ***
  7|            ***  ***
  6|            ***  ***  ***
  5|       ***  ***  ***  ***
  4|  ***  ***  ***  ***  ***
  3|  ***  ***  ***  ***  ***  ***
  2|  ***  ***  ***  ***  ***  ***  ***
  1|  ***  ***  ***  ***  ***  ***  ***

   ________________________________________
~/PythonBeginner/Lesson7 $ python charts.py
 10|                    ****
  9|                    ****
  8|                    ****
  7|              ****  ****
  6|              ****  ****  ****
  5|        ****  ****  ****  ****
  4|  ****  ****  ****  ****  ****
  3|  ****  ****  ****  ****  ****  ****
  2|  ****  ****  ****  ****  ****  ****  ****
```

100% z Lekce 11

Osnova

Error Handling Tic Tac Toe [H]

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / HOME EXERCISES / ERROR HANDLING TIC TAC TOE [H]

You had the chance to created the game Tic Tac Toe. This games probably works perfectly, if all the inputs from the user are as expected by the programmer. However, this is not the real-life scenario. Users may write their input incorrectly and thus cause program to crash.

Your task is to review your Tic Tac Toe app and put exception handling statements, where appropriate. Your task is to **discover all the weak spots and fix them**.

Example of errors that could be handled:

```
~/PythonBeginner/Lesson7 $ python tic_tac.py
    ===========================
    Welcome to Tic Tac Toe
    GAME RULES:
    Each player can place one mark (or stone) per turn on the 3x3 grid
    The WINNER is who succeeds in placing three of their marks in a
    * horizontal,
    * vertical or
    * diagonal row
    Let's start the game
Please enter the board size: s
Traceback (most recent call last):
  File "tic_tac.py", line 107, in <module>
    main()
  File "tic_tac.py", line 91, in main
    board, player = setup_game()
  File "tic_tac.py", line 32, in setup_game
    size = int(input('Please enter the board size: '))
```

100% z Lekce 11

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Function setup_game()

This function is the first one that takes user input and thus is susceptible to introduction of errors. If the user does not enter a whole number, we will have problems converting this input into an integer data type.

The other problem may arise, if the input value is less than 1 - in that case the game is not possible. We decided to create custom error called `BoardSizeError`, which will be raised in this case.

```
def setup_game():
    # handled 1
    user_input = input('Please enter the board size: ')

    if not user_input.isdecimal():
        raise ValueError('Only whole numbers are permitted\n')
    elif int(user_input) < 2:
        raise BoardSizeError('The board size has to be at least 2\n')

    size = int(user_input)
    board = []
    for row in range(size):
        board.append([' '] * size)
    player_to_start = random.choice(['x','o'])

    return board, player_to_start
```

Ar… …ion th…t takes input from the user. It records the player's move on the board.
Osnova
I… …a… …oes enter the expected value - integer between 1 and board size raised to power two, the error occurs.

Therefore we have implemented check, that raises `IncorrectMoveError` if the move entered by the user is not a whole integer between 1 and size raised to two. This exception is later caught inside the `main()` function. Note that we send the message to be printed to the terminal with the error (passing it inside the parentheses).

```
def make_move(player,board):
    size = len(board)

    while True:

        move = input('Player {} | Please enter your move number: '.format(player))

        if not move.isdecimal() or not 1<=int(move)<= size**2:
            raise IncorrectMoveError('Please enter only numbers between 1 - - %d'%(size**2))

        move = int(move)

        row,col =  to_coord(move, len(board))
        if board[row][col] == ' ':
            board[row][col] = player
            break
        else:
            print('\nThis position is already taken\n')
    return board
```

Function main()

As two functions called within the `main()` function raise errors, we need to handle them somehow. `setup_game()` raises two kinds of errors `BoardSizeError` and `ValueError`. Both errors contain message may want to print to inform the user. This message is carried inside the `.args` attribute of the error object.

100% z Lekce 11

th[illegible]ies. Also note, we have included both errors in one `except` clause.

Basically the same thing is performed with the `make_move()` function.

```
def main():
    print_intro()

    while True:
        try:
            board, player = setup_game()
            break
        except (BoardSizeError, ValueError) as e:
            print(e.args[0])
            continue

    while True:
        print_board(board)

        try:
            board = make_move(player, board)
        except IncorrectMoveError as e:
            print(e.args[0])
            continue

        if check_game_over(board,player):
            print_game_over(player)
            break
        elif board_full(board):
            print_tie()
            break

        player = swap_players(player)
    print_board(board)
```

After putting in place the above controls, our code handles incorrect user inputs smoothly:

```
$ python tic_tac_handled.py
    ===========================
    Welcome to Tic Tac Toe
```

```
    The WINNER is who succeeds in placing three of their marks in a
    Osnova ( ,
    * vertical or
    * diagonal row
    Let's start the game
Please enter the board size: abc
Only whole numbers are permitted
Please enter the board size: 2.5
Only whole numbers are permitted
Please enter the board size: 3
------
 | |
------
 | |
------
 | |
------
Player o | Please enter your move number: abc
Please enter only numbers between 1 - 9
------
 | |
------
 | |
------
 | |
------
Player o | Please enter your move number: 1
------
o| |
------
 | |
------
 | |
------
```

100% z Lekce 11

Osnova

Tac Toe [H]

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / HOME EXERCISES / LOGGING TIC TAC TOE [H]

Create your own logger function that will log all the errors that will be raised in the application into a file called `tic_tac_log.txt` .

Your logger should record the following information:

- **filename**
- **function name**
- **line number**
- **error type and error message**

Bonus

Try to record also the value, the user entered.

Example how such log file could look like:

```
REPORT
FILE tic_tac_handled.py / FUNCTION: main / LINE 113 / VALUE: 1
FILE tic_tac_handled.py / FUNCTION: setup_game / LINE 40 / VALUE: 1
BoardSizeError: The board size has to be at least 2


REPORT
FILE tic_tac_handled.py / FUNCTION: main / LINE 124 / VALUE: 6534
FILE tic_tac_handled.py / FUNCTION: make_move / LINE 56 / VALUE: 6534
IncorrectMoveError: Please enter only numbers between 1 - 9

REPORT
FILE tic_tac_handled.py / FUNCTION: main / LINE 124 / VALUE: `
FILE tic_tac_handled.py / FUNCTION: make_move / LINE 56 / VALUE: `
IncorrectMoveError: Please enter only numbers between 1 - 9
```

100% z Lekce 11

Osnova

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Function main()

Our `main()` function begins to grow as we keep adding more and more functionality. In order our program to log the errors that have been raised, we need to put the `logger()` function inside the `except` clauses.

Also note, we have enclosed almost the whole `main()` code inside a context manager, just in case the program is terminated unexpectedly and the file can be automatically closed.

```
def main():
    print_intro()

    with open('tic_tac_log.txt','a') as log_file:


        while True:
            try:
                board, player = setup_game()
                break
            except (BoardSizeError, ValueError) as e:
                print(e.args[0])
                logger(sys.exc_info(), file=log_file)
                continue


        while True:
            print_board(board)

```

100% z Lekce 11

Osnova

```
            print(e.args[0])
            logger(sys.exc_info(), file=log_file)
            continue

        if check_game_over(board,player):
            print_game_over(player)
            break
        elif board_full(board):
            print_tie()
            break

        player = swap_players(player)

    print_board(board)
```

Function logger()

Our logging function makes use of traceback module's `extract_tb()` function. It extracts filename, function name etc. from individual FrameSummary objects. Also note, that we have unpacked the "message" that represents the error object into two separate variables `msg` and `input_value` . The first got the message we have passed to the error object and the latter received the value that the user has entered.

```
def logger(exception_info,*, file=sys.stdout):
    error, err_obj, tb = exception_info

    msg, inpt_value = err_obj.args

    summary = traceback.extract_tb(tb)

    report = []
    for s in summary:

        report.append('FILE {} / FUNCTION: {} / LINE {} / VALUE: {}'
.format(s.filename,s.name,s.lineno,inpt_value))

    report.append('{__name__}: {}\n'.format(error, msg))
```

100% z Lekce 11

tup_game() and make_move()

To include values that cause the error, we can send them as argument in the error object. In the `setup_game`, we store the input inside `user_input`. In the `make_move()` we store the input inside `move` variable.

```
def setup_game():

    user_input = input('Please enter the board size: ')

    if not user_input.isdecimal():
        raise ValueError('Only whole numbers are permitted\n',user_input)
    elif int(user_input) < 2:
        raise BoardSizeError('The board size has to be at least 2\n',user_input)

    size = int(user_input)
    board = []
    for row in range(size):
        board.append([' '] * size)
    player_to_start = random.choice(['x','o'])

    return board, player_to_start
```

```
def make_move(player,board):

    size = len(board)

    while True:

        move = input('Player {} | Please enter your move number: '.format(player))

        if not move.isdecimal() or not 1<=int(move)<= size**2:
            raise IncorrectMoveError('Please enter only numbers between 1 - %d'%(size**2),move)

```

```
        row,col =  to_coord(move, len(board))
```



```
        if board[row][col] == ' ':
            board[row][col] = player
            break
        else:
            print('\nThis position is already taken\n')

    return board
```

DALŠÍ LEKCE