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.

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

Using string formatting to create nicely formatted output table.

Example of program in use:

```
~/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

Error Handling

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:
    handled exception
```

Example:

```
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):
    print('Enter numbers only')
```

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()
```

Usually context manager used instead of **try-finally**

```
with open(filename) as file:
    file.write(content)
```

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:

```
raise ExceptionName
```

```
raise NameError('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
- `tb_next` - next inner traceback object (called by this level)

We would need to use `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
- `lineno` - line number, where the error occured
- `name` - name of the function - execution frame - where the error occured

This may be the most complete source for our custom logging solution.

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 control.

Python's solution to this requirement is raising of so called **exceptions** when an error occurs. Once the exception is 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

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 declared, and thus does not exist. In other words, we are using a name, that does not exist. Therefore the exception `NameError` is raised. This is documented using so called **traceback**:

```
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:

```
>>> 8/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: 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 True:
... print('Yes')
  File "<stdin>", line 2
    print('Yes')
        ^
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'
```

To avoid the `KeyError` to be raised, we can use dictionary's `get()` method for key lookup:

```
>>> d.get('age','No such key')
'No such key'
```

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

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:
...     print('Please enter numbers only')
...
Please enter a number: abc
Please enter numbers 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
Please enter numbers only
```

Why to specify a concrete error name? There are multiple reasons why we should specify the name of the error we would like to handle.

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.

The other reason to include error name in except clause is that we maybe want to **catch only some exceptions and 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

There are many exceptions and some of them pertain to the same family (group). Some exceptions are more 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`.

```
>>> try:
...     a = nonexistent * 5
... except BaseException:
...     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

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / 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:
   statements
```

Another often used 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

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

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

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
except:
    exception raised -> statements executed
else:
    no exception 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

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / ERROR HANDLING / 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:
    print('Could not find the file')
else:
    data = csv.reader(fl).read()
```

```
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.

B. If an **exception does not occur** while the try block's statements are running,

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.

Raising Exceptions

We raise exceptions on purpose by using the raise statement:

```
raise ExceptionName
```

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:

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

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

An example would be an exception raised every time a player playing chess game would try to perform illegal movement:

```
class IncorrectMove(Exception):
    pass
```

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

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.

```
1  def count(sequence, target=None):
2      if not target:
3          return len(sequence)
4      else:
5          count = 0
6          for item in sequence:
7              count += item==target
8          return count
9
10
11 def my_sum(sequence):
12     result = 0
13     for i in sequence:
14             result += i
15     return result
16
17 def avg(sequence):
18     try:
19         return my_sum(sequence) / count(sequence)
20     except ZeroDivisionError:
21         print('Zero Division Error has been handled')
22     except TypeError:
23         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

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
... except TypeError:
...     2 / 0
...
Traceback (most recent 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 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

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.

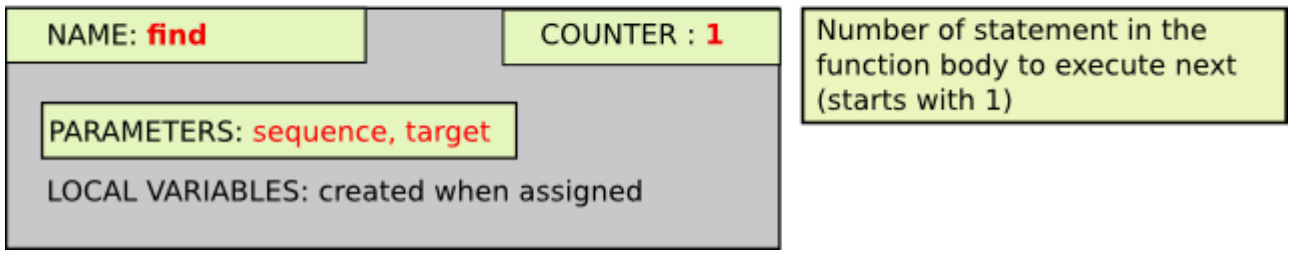
We can imagine execution frame as a **box**, that keeps track of:

- function name,
- execution counter - what line is being currently executed,
- local and global variables.

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

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
```



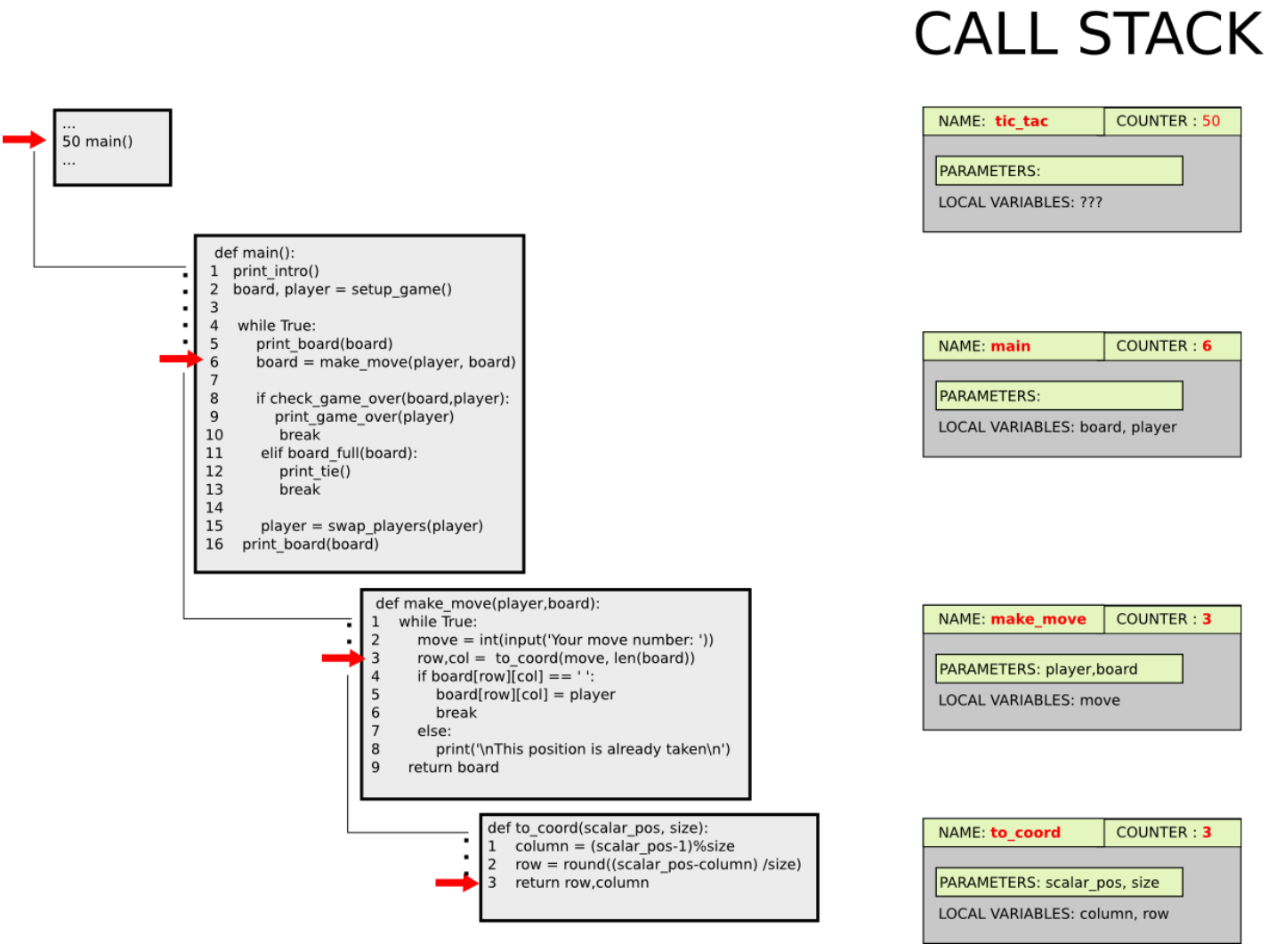
There are many other Python concepts, that when run, execution frame is created. We will however not mention them here.

Call Stack

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.

If you are asking, why do we need to know, what is execution frame and call stack, the answer is: **to understand program execution and traceback reading**.



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` :

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

We would get the following error report:

```
Lesson7 $ python load_db.py
Traceback (most recent call last):
  File "load_db.py", line 1, in <module>
    file = open('nonexistent.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent.txt'
```

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):
    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)
```

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 the error exactly to its beginning: `File "<stdin>", line 4, in my_sum` - line 4 in the function `my_sum` .

Solving the Bug

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 :

```
1  def count(sequence, target=None):
2      if not target:
3          return len(sequence)
4      else:
5          count = 0
6          for item in sequence:
7              count += item==target
8          return count
9
10
11 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
```

Assert Statement

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING / ASSERT STATEMENT

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:

```
1  def avg(sequence):
2      assert len(sequence) > 0, 'Cannot accept empty sequences'
3      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

In particular, they're good for catching false assumptions that were made while writing the code, or incorrect use of our 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

Accessing Exception Information

We can access information about the last error in the following ways:

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

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
...
```

We want to store the content of the variable `e` in another variable (`err`), because once the program execution leaves the `except` block, variable `e` is destroyed.

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

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

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:

```
1  def count(sequence, target=None):
2      if not target:
3          return len(sequence)
4      else:
5          count = 0
6          for item in sequence:
7              count += item==target
8          return count
9
10
11 def my_sum(sequence):
12     result = 0
13     for i in sequence:
14             result += i
15     return result
16
17 def avg(sequence):
18     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 run further queries using `tb_next`, we get all the lines from the traceback until we arrive at line 4 in `my_sum`:

```
>>> 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'
<traceback object at 0x7f9bad46eb08>
```

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

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>]
```

`FrameSummary` object reflect the information about a single execution frame. From the above error report, we can see, 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 `sys.exc_info()` function to extract all the information about the traceback and the error. Code that is supposed to extract all this information is placed inside the `except` clause, inside function `avg` :

```
1  import traceback,sys
2
3  def count(sequence, target=None):
4      if not target:
5          return len(sequence)
6      else:
7          count = 0
8          for item in sequence:
9              count += item==target
10     return count
11
12
13 def my_sum(sequence):
14     result = 0
15     for i in sequence:
16         result += i
17     return result
18
19 def avg(sequence):
20     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'])
```

Running the above script leaves us with the following error summary:

```
~/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 test_traceback.py / FUNCTION: avg / LINE 34:
    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'
```

Stream Redirection

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / 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):
```

```
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:

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

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

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

PYTHON ACADEMY / 9. ERRORS & DEBUGGING / DEBUGGING + / ADDITIONAL MODULES

As we have mentioned at the beginning of this section, Python has its additional modules that are useful error handling & 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.

Error Handling & Debugging

1/19

What is an **exception**?

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.

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
```

spustit kód

Code Solution

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

Click to see our solution

Click to see our solution

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
IX	9
XL	40
XC	90
CD	400

Roman digits	Arabic Equivalent
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

Horizontal Histogram

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

Create a very simple function, that will print the input list of numbers as a histogram. For example, if the function receives the following values `[4,5,7,10,6,3,2]` it should print seven lines - one for each item. Each line should contain as many stars `*` as is the magnitude of the number (number 4 -> 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
```

spustit kód

Code Solution

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

Click to see our solution

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|              **
  8|              **
  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|  ****  ****  ****  ****  ****
  3|  ****  ****  ****  ****  ****  ****
  2|  ****  ****  ****  ****  ****  ****  ****
  1|  ****  ****  ****  ****  ****  ****  ****
   ____________________________________________
```

Code Solution

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

Click to see our solution

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

Code Solution

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

Click to see our solution

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:

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

```
5
6
7  REPORT
8  FILE tic_tac_handled.py / FUNCTION: main / LINE 124 / VALUE: 6534
9  FILE tic_tac_handled.py / FUNCTION: make_move / LINE 56 / VALUE: 6534
10 IncorrectMoveError: Please enter only numbers between 1 - 9
11
12 REPORT
13 FILE tic_tac_handled.py / FUNCTION: main / LINE 124 / VALUE: `
14 FILE tic_tac_handled.py / FUNCTION: make_move / LINE 56 / VALUE: `
15 IncorrectMoveError: Please enter only numbers between 1 - 9
```

Code Solution

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

Click to see our solution

DALŠÍ LEKCE