SNS COLLEGE OF ENGINEERING
Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF CSE (IoT & CYBER SECURITY INCLUDING BLOCKCHAIN TECHNOLOGY)

19IT103 – COMPUTATIONAL THINKING AND PYTHON PROGRAMMING

❖ **A readable, dynamic, pleasant, flexible, fast and powerful language**

Objective

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file, Voter’s age validation, Marks range validation (0-100).

RECAP

A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)

Python Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:

- **Exception Handling:** This would be covered in this session.
- **Assertions:** This would be covered in Assertions in Python.

What is Exception?

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Handling an exception:

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:

```
try:
   You do your operations here;
   ......................
except Exception I:
   If there is ExceptionI, then execute this block.
except Exception II:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points above the above mentioned syntax:

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception
   handling!!")
except IOError: print "Error: can\'t find file or read
   data"
else: print "Written content in the file successfully"
fh.close()
```

- This will produce following result:

```
Written content in the file successfully
```

The *except* clause with no exceptions:

You can also use the except statement with no exceptions defined as follows:

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice, though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* clause with multiple exceptions:

You can also use the same *except* statement to handle multiple exceptions as follows:

```
try:
  You do your operations here;
  ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
  If there is any exception from the given exception
  list, then execute this block
  .......................
else:
  If there is no exception then execute this block.
```

Standard Exceptions:

Here is a list standard Exceptions available in Python: Standard Exceptions

The try-finally clause:

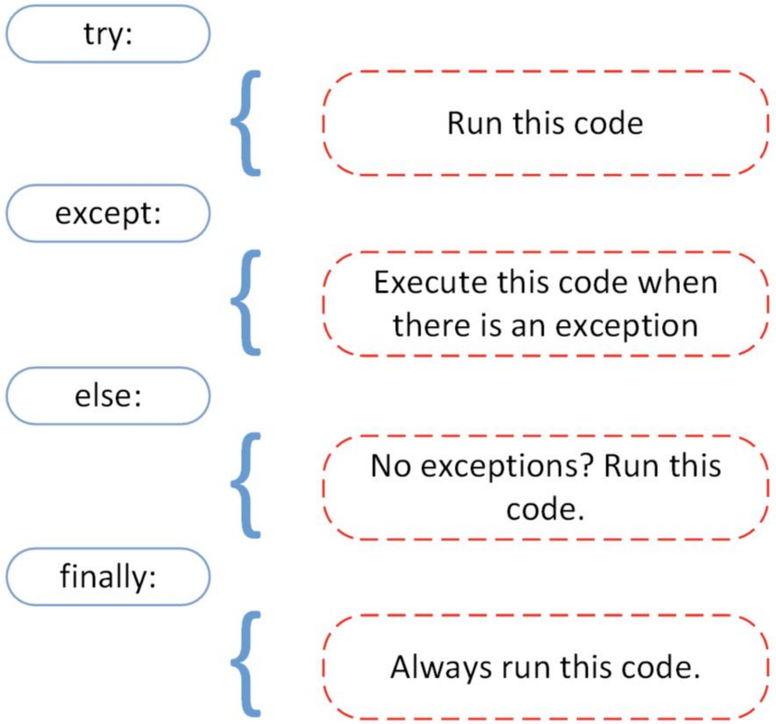
You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

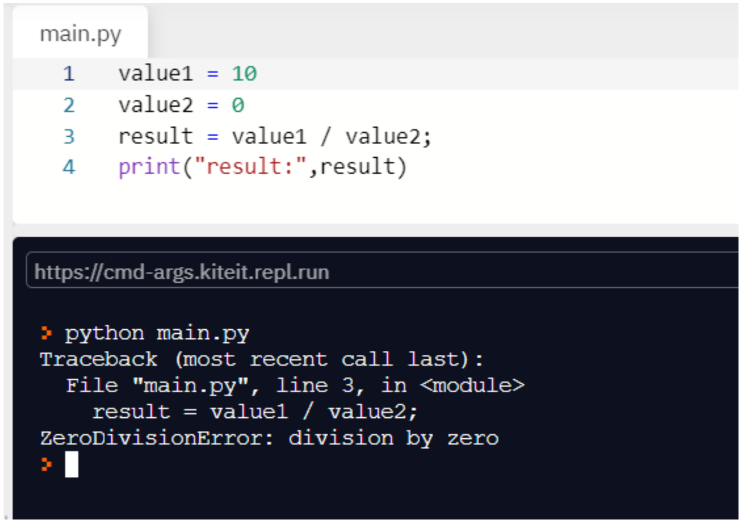
Note that you can provide except clause(s), or a finally clause, but not both. You can not use *else* clause as well along with a finally clause.

Handling Exceptions

Exceptions are handled by special blocks


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Before and After (Handling)

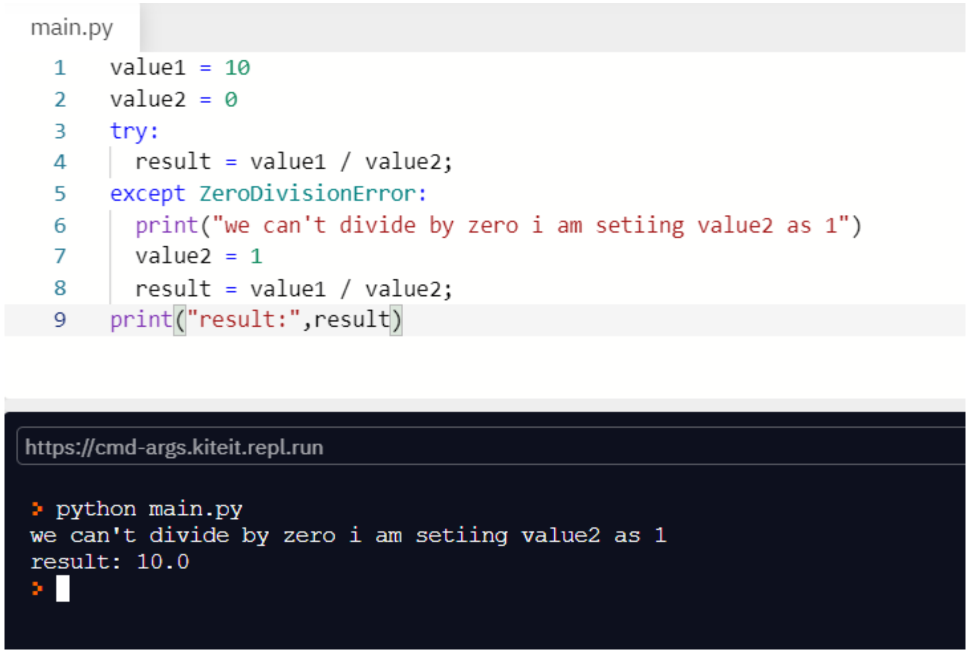


main.py

```python
value1 = 10
value2 = 0
result = value1 / value2;
print("result:",result)
```

https://cmd-args.kiteit.repl.run

```
> python main.py
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    result = value1 / value2;
ZeroDivisionError: division by zero
>
```



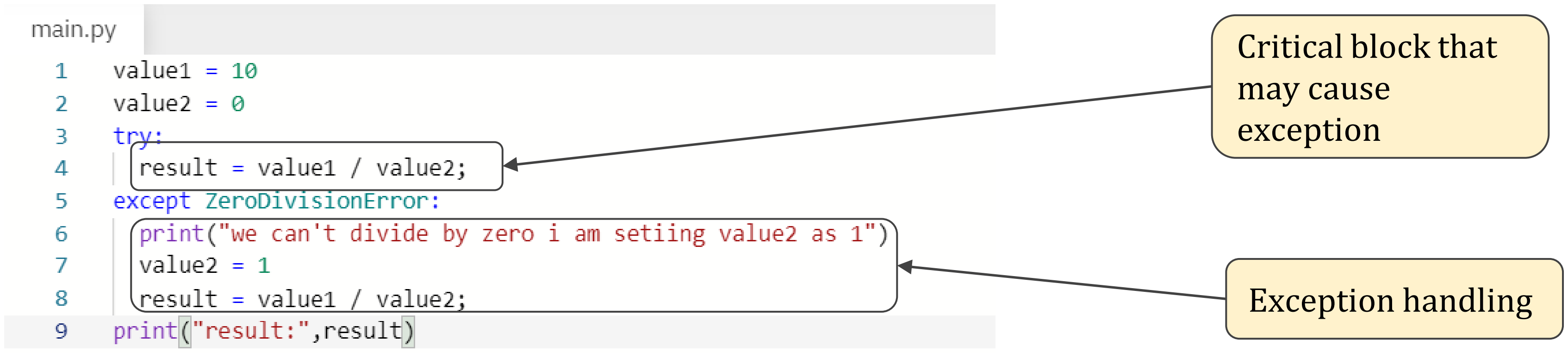
main.py

```python
value1 = 10
value2 = 0
try:
  result = value1 / value2;
except ZeroDivisionError:
  print("we can't divide by zero i am setiing value2 as 1")
  value2 = 1
  result = value1 / value2;
print("result:",result)
```

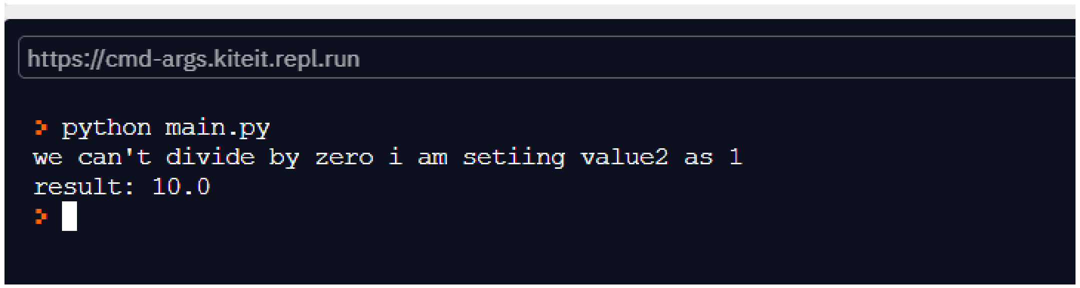
https://cmd-args.kiteit.repl.run

```
> python main.py
we can't divide by zero i am setiing value2 as 1
result: 10.0
>
```

Before and After (Handling)

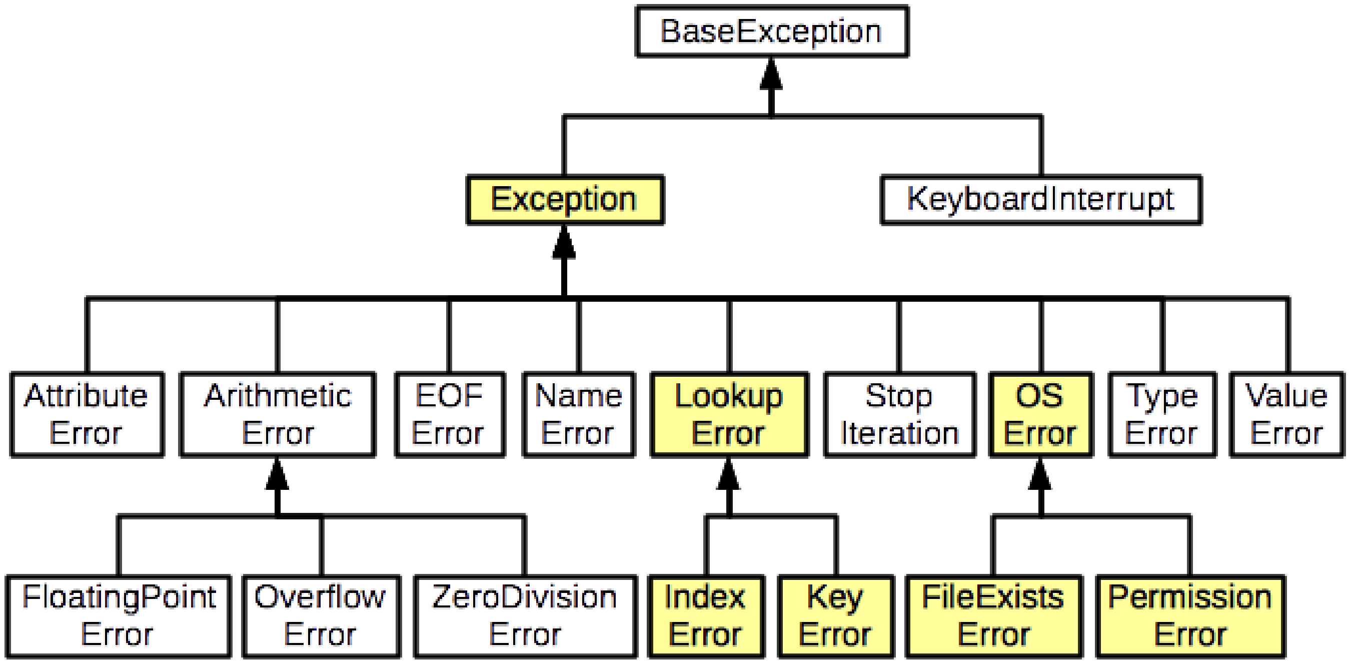

main.py

```python
value1 = 10
value2 = 0
try:
    result = value1 / value2;
except ZeroDivisionError:
    print("we can't divide by zero i am setiing value2 as 1")
    value2 = 1
    result = value1 / value2;
print("result:",result)
```

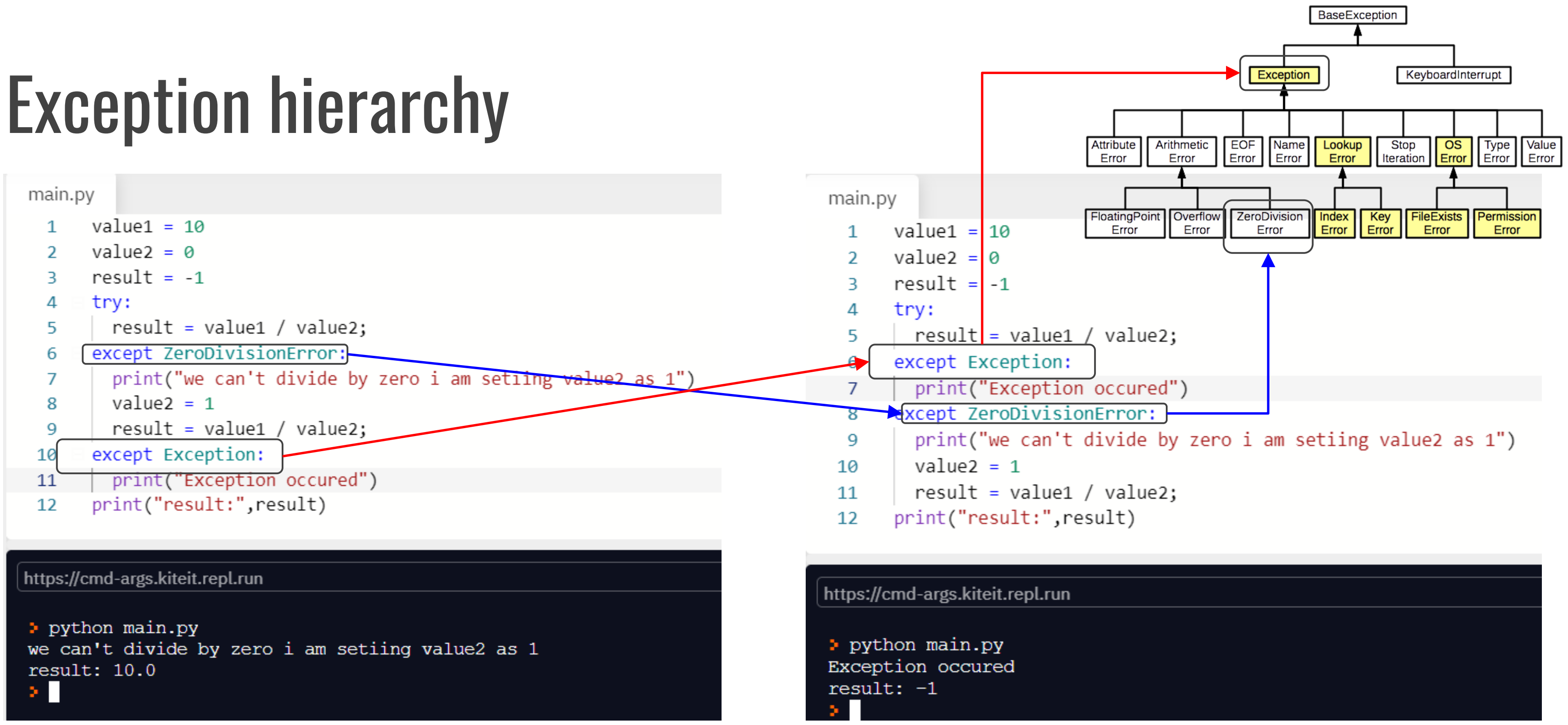

https://cmd-args.kiteit.repl.run

```
> python main.py
we can't divide by zero i am setiing value2 as 1
result: 10.0
>
```

Python Exception Hierarchy



Exception hierarchy



main.py

```python
value1 = 10
value2 = 0
result = -1
try:
  result = value1 / value2;
except ZeroDivisionError:
  print("we can't divide by zero i am setiing value2 as 1")
  value2 = 1
  result = value1 / value2;
except Exception:
  print("Exception occured")
print("result:",result)
```

```
https://cmd-args.kiteit.repl.run

> python main.py
we can't divide by zero i am setiing value2 as 1
result: 10.0
>
```

main.py

```python
value1 = 10
value2 = 0
result = -1
try:
  result = value1 / value2;
except Exception:
  print("Exception occured")
except ZeroDivisionError:
  print("we can't divide by zero i am setiing value2 as 1")
  value2 = 1
  result = value1 / value2;
print("result:",result)
```

```
https://cmd-args.kiteit.repl.run

> python main.py
Exception occured
result: -1
>
```

Priority Matters

Raising an exceptions:

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement.

Syntax:

```
raise [Exception [, args [, traceback]]]
```

- Here *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.
- The final argument, traceback, is also optional (and rarely used in practice), and, if present, is the traceback object used for the exception

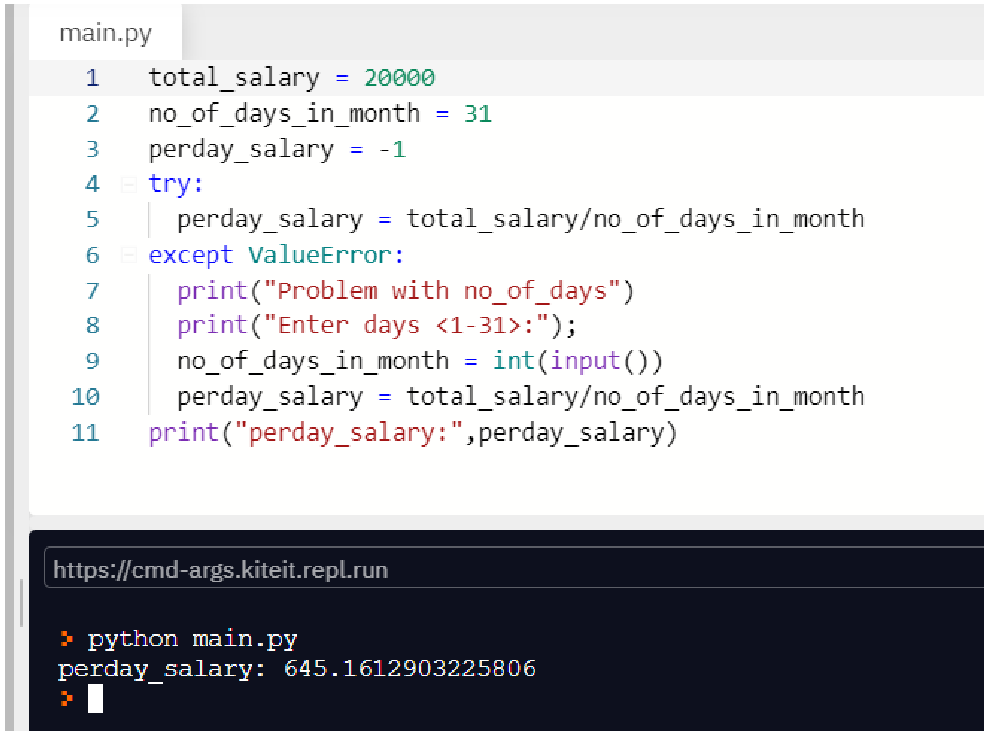
Example:

```
def functionName( level ):
   if level < 1:
   raise "Invalid level!", level
   # The code below to this would not be executed
   # if we raise the exception
```

Raising Exception (normal case)

We can manually raise the exception using raise keyword.

Example - Compute per day salary for a month



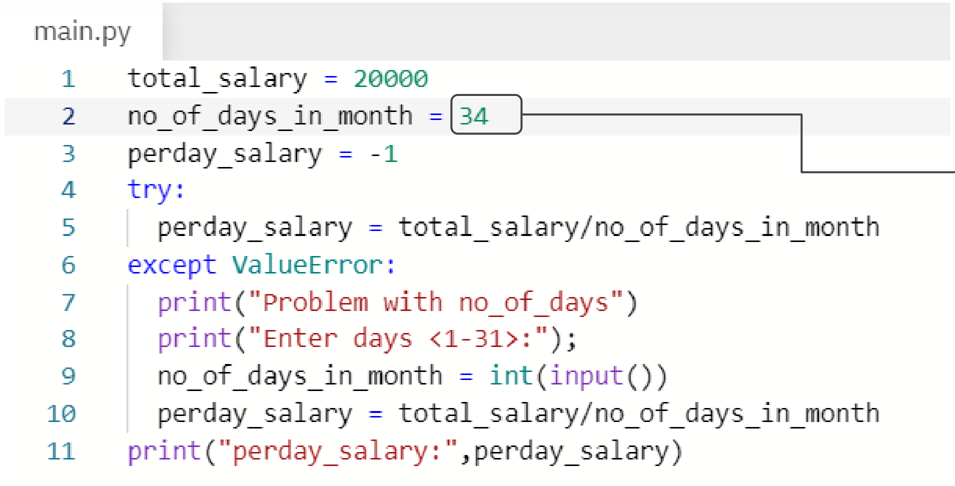
main.py

```python
total_salary = 20000
no_of_days_in_month = 31
perday_salary = -1
try:
    perday_salary = total_salary/no_of_days_in_month
except ValueError:
    print("Problem with no_of_days")
    print("Enter days <1-31>:");
    no_of_days_in_month = int(input())
    perday_salary = total_salary/no_of_days_in_month
print("perday_salary:",perday_salary)
```

https://cmd-args.kiteit.repl.run

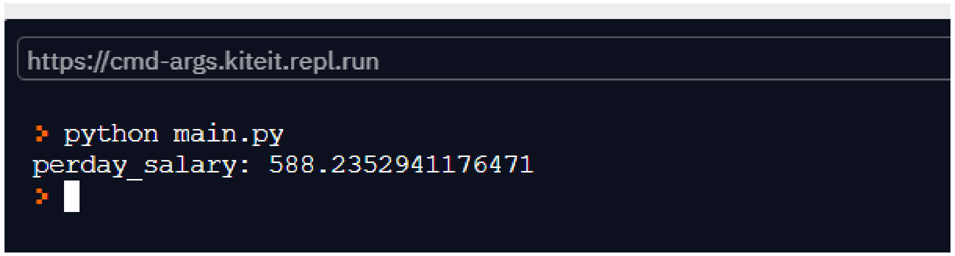
```
> python main.py
perday_salary: 645.1612903225806
>
```

Problem



main.py

```python
total_salary = 20000
no_of_days_in_month = 34
perday_salary = -1
try:
    perday_salary = total_salary/no_of_days_in_month
except ValueError:
    print("Problem with no_of_days")
    print("Enter days <1-31>:");
    no_of_days_in_month = int(input())
    perday_salary = total_salary/no_of_days_in_month
print("perday_salary:",perday_salary)
```



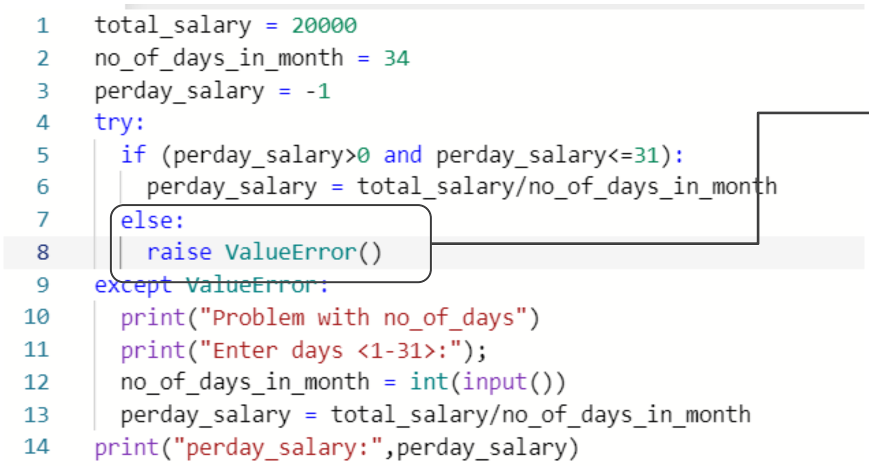
https://cmd-args.kiteit.repl.run

```
> python main.py
perday_salary: 588.2352941176471
> 
```

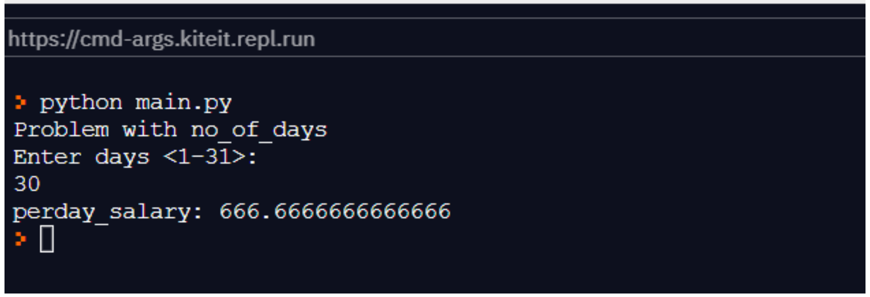
Not acceptable but

- No errors (syntactically correct)
- No Exceptions (Of Course we can divide by 34)

Solution



```python
total_salary = 20000
no_of_days_in_month = 34
perday_salary = -1
try:
  if (perday_salary>0 and perday_salary<=31):
    perday_salary = total_salary/no_of_days_in_month
  else:
    raise ValueError()
except ValueError:
  print("Problem with no_of_days")
  print("Enter days <1-31>:");
  no_of_days_in_month = int(input())
  perday_salary = total_salary/no_of_days_in_month
print("perday_salary:",perday_salary)
```



https://cmd-args.kiteit.repl.run

```
> python main.py
Problem with no_of_days
Enter days <1-31>:
30
perday_salary: 666.6666666666666
> 
```

- Manually raising exception

User-Defined Exceptions:

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.
- Here is an example related to *RuntimeError*. Here a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.
- In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
  def __init__(self, arg):
    self.args = arg
```

- So once you defined above class, you can raise your exception as follows:

```
try:
  raise Networkerror("Bad hostname")
except Networkerror,e:
  print e.args
```

SUMMARY

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- Handling Exceptions
- Raising an exceptions
- User-Defined Exceptions

Thank you