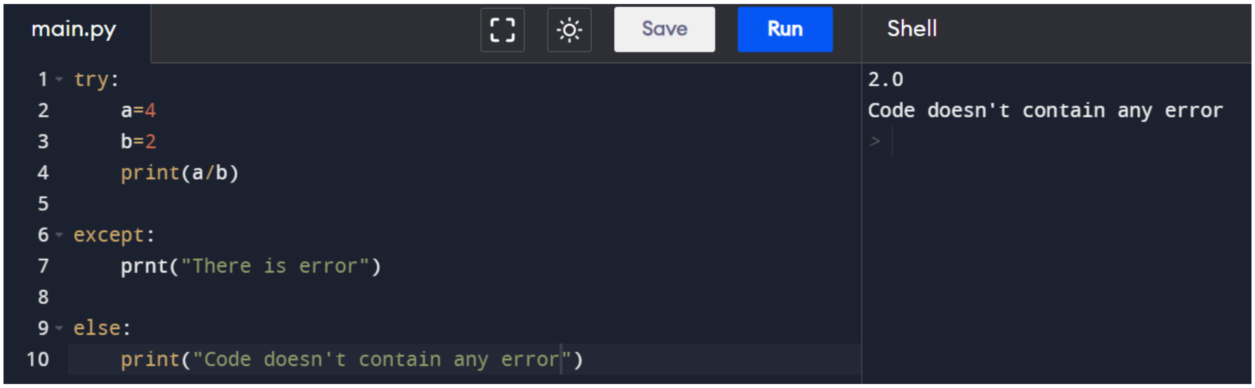
Exceptions and Logging

1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.

ANSWER: The try block lets you test a block of code for errors. The except block lets you handle the error. The else block lets you execute code when there is no error.

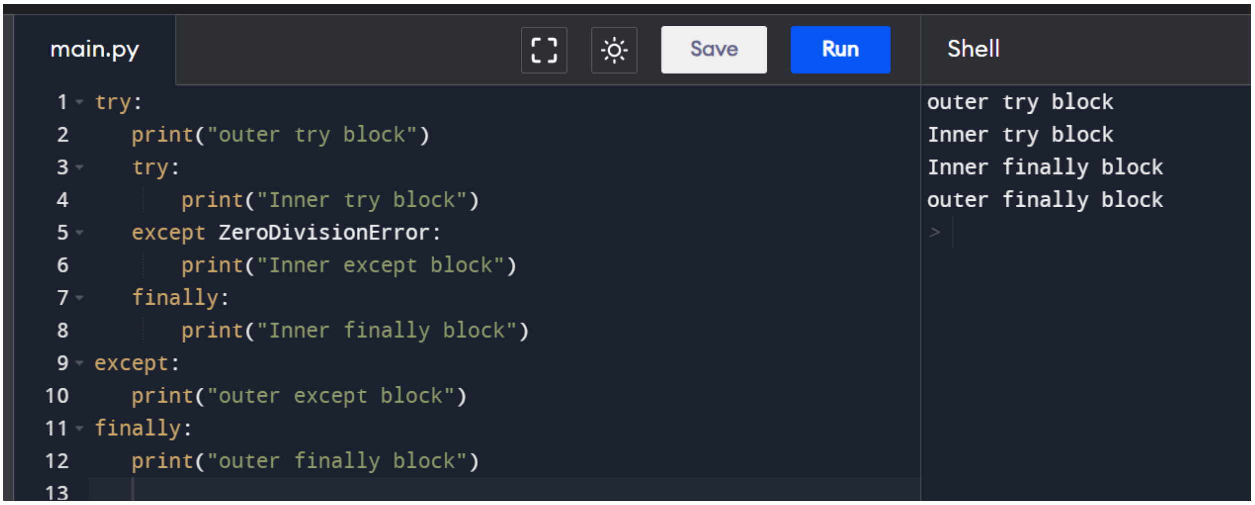


```python
try:
    a=4
    b=2
    print(a/b)

except:
    prnt("There is error")

else:
    print("Code doesn't contain any error")
```

```
2.0
Code doesn't contain any error
```

2. Can a try-except block be nested inside another try-except block? Explain with an example.

ANSWER: We can have nested try-except blocks in Python. In this case, if an exception is raised in the nested try block, the nested except block is used to handle it. In case the nested except is not able to handle it, the outer except blocks are used to handle the exception.

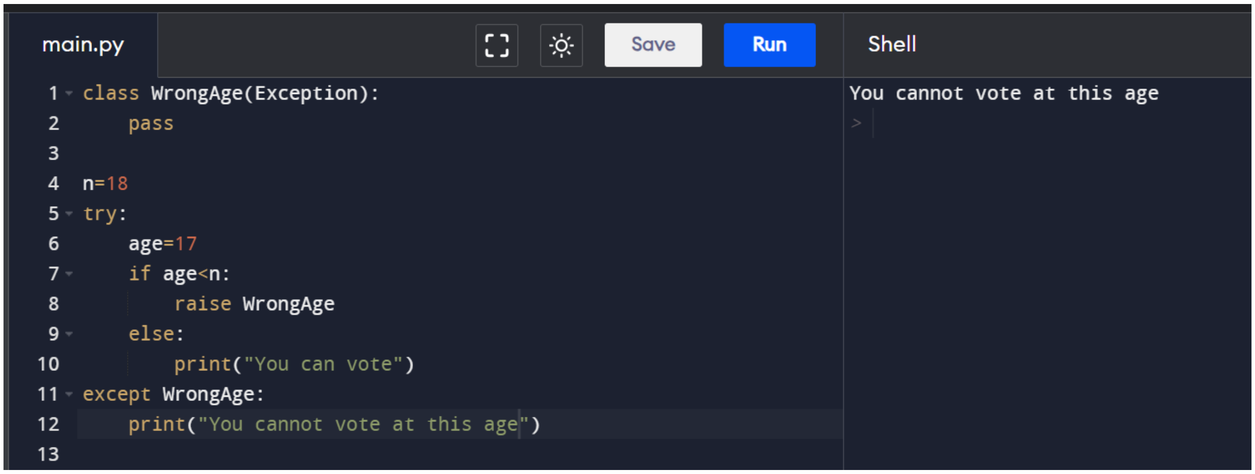


```python
try:
    print("outer try block")
    try:
        print("Inner try block")
    except ZeroDivisionError:
        print("Inner except block")
    finally:
        print("Inner finally block")
except:
    print("outer except block")
finally:
    print("outer finally block")
```

```
outer try block
Inner try block
Inner finally block
outer finally block
```

3. How can you create a custom exception class in Python? Provide an example that demonstrates its usage.

ANSWER: In Python, you can create a custom exception by defining a new class that inherits from the built-in `Exception` class or one of its subclasses. To raise the custom exception, you can use the `raise` keyword followed by an instance of your custom exception class.



```
class WrongAge(Exception):
    pass

n=18
try:
    age=17
    if age<n:
        raise WrongAge
    else:
        print("You can vote")
except WrongAge:
    print("You cannot vote at this age")

```

```
You cannot vote at this age
>
```

4. What are some common exceptions that are built-in to Python?

ANSWER: ZeroDivisionError, TypeError, ValueError, EOFError, FloatingPointError, ImportError, IndexError, ModuleNotFoundError, MemoryError, NameError etc.

5. What is logging in Python, and why is it important in software development?

ANSWER: Logging is a way to store information about your script and track events that occur. When writing any complex script in Python, logging is essential for debugging software as you develop it. Without logging, finding the source of a problem in your code may be extremely time consuming.

6. **Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.**

ANSWER: Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the *level* or *severity*.

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it’s used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

7. What are log formatters in Python logging, and how can you customise the log message format using formatters?

ANSWER: A formatter is created and added to the handler to transform log messages into placeholder data. In this formatter, the time of the log request (as an epoch timestamp), the logging level, the logger's name, the module name, and the log message will all print.

Below are the steps for creating custom logger.

Create a logger:

```
logger = logging.getLogger(‘demologger’)
logger.setLevel(logging.INFO)
```

Creating handler:

For Stream Handler,

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.INFO)
```

For File Handler,

```
fileHandler = logging.FileHandler(‘test.log’)
fileHandler.setLevel(logging.INFO)
```

Creating Formatter:

```
formatter = logging.Formatter(‘%(asctime)s – %(name)s – %(levelname)s: %(message)s’, datefmt=’%d/%m/%Y %I:%M:%S %p’)
```

Adding Formatter to Handler:

```
consoleHandler.setFormatter(formatter)
```

Adding Handler object to the Logger:

```
logger.addHandler(fileHandler)
logger.addHandler(streamHandler)
```

Writing the log messages:

```
logger.debug(‘debug message’)
logger.info(‘info message’)
logger.warn(‘warn message’)
logger.error(‘error message’)
logger.critical(‘critical message’)
```

8. How can you set up logging to capture log messages from multiple modules or classes in a Python application?

ANSWER: For that, simply import the module from the library.

1. Create and configure the logger. ...
2. Here the format of the logger can also be set. ...
3. Also, the level of the logger can be set which acts as the threshold for tracking based on the numeric values assigned to each level.

9. What is the difference between the logging and print statements in Python? When should you use logging over print statements in a real-world application?

ANSWER:

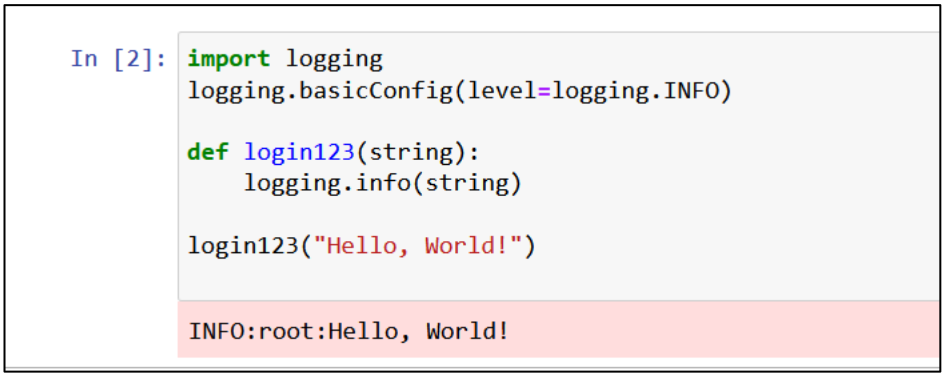
Logging in Python	Print in Python
Record events and errors that occur during the execution of Python programs.	Displays the information to the console for the debugging purposes.
Mainly used in the production environment.	Mainly for debugging.
Some features are: Log levels, filtering, formatting, and more.	There are no good features.
It provides different log levels such as Debug, Info, Error, Warning, and Critical.	It does not have any levels, it simply prints whatever is passed to it.
Example: import logging; logging.basicConfig(level=logging.INFO); logging.info(“Hello”) **Output:** Can be configured to log to different output destinations (e.g. console, file, network)	Example: print(“Hello”) **Output:** Prints only on the console

- For big projects logging is always a "best practice" because you can easily turn it on or off, and get more or less information. print offers neither of these advantages.

10. Write a Python program that logs a message to a file named "app.log" with the following requirements:

- **The log message should be "Hello, World!"**
- **The log level should be set to "INFO."**
- **The log file should append new log entries without overwriting previous ones.**

ANSWER:



```
import logging
logging.basicConfig(level=logging.INFO)

def login123(string):
    logging.info(string)

login123("Hello, World!")
```

```
INFO:root:Hello, World!
```