Programming in Python 1

Errors, Exceptions, Assertions, Strings

MSc Paweł Pięta

Kielce University of Technology
Faculty of Electrical Engineering, Automatic Control and Computer Science
Department of Information Systems: Division of Computer Science

December 3, 2018

Plan of the Presentation

Plan of the Presentation

Python Syntax Errors and Exceptions

There are two distinguishable kinds of errors:

- syntax errors,
- exceptions.

Python Syntax Errors

- Syntax errors, also known as *parsing errors*, are perhaps the most common kind of complaint you get while you are still learning Python.
- When the parser encounters a syntax error, it prints the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected.
- The error is caused by (or at least detected at) the token preceding the arrow.
- File name and line number are also printed so you know where to look in case the input came from a script.

Python Syntax Error – Example

```
1 while True print('Hello world')
```

Output

```
  File "test.py", line 1
    while True print('Hello world')
                     ^
SyntaxError: invalid syntax
```

Python Syntax Error – Another Example

```
1 def func():
2     global var = 1
3     print(var)
```

Output

```
  File "test.py", line 2
    global var = 1
               ^
SyntaxError: invalid syntax
```

Python Syntax Error – Yet Another Example

```
1 def introduction(firstname="John", lastname):
2     print("My name is", firstname, lastname, end=".\n")
```

Output

```
  File "test.py", line 1
    def introduction(firstname="John", lastname):
                     ^
SyntaxError: non-default argument follows default argument
```

What is an Exception?

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

- Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions.
- An exception is raised at the point where the error is detected.
- It may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

How Python Exceptions are Raised?

1. The Python interpreter raises an exception when it detects a run-time error (such as division by zero).
2. A Python program can also explicitly raise an exception with the `raise` statement.

Examples of Exceptions

```
1 >>> 10 * (1/0)
```

Output

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10 * (1/0)
ZeroDivisionError: division by zero
```

```
1 >>> 4 + spam*3
```

Output

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 + spam*3
NameError: name 'spam' is not defined
```

Examples of Exceptions – Continued

```
1 >>> '2' + 2
```

Output

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '2' + 2
TypeError: can only concatenate str (not "int") to str
```

```
1 >>> x = int(input("Enter number: "))
```

Output (user input: "text")

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    x = int(input("Enter number: "))
ValueError: invalid literal for int() with base 10: 'text'
```

Hierarchy of 64 Built-in Exceptions

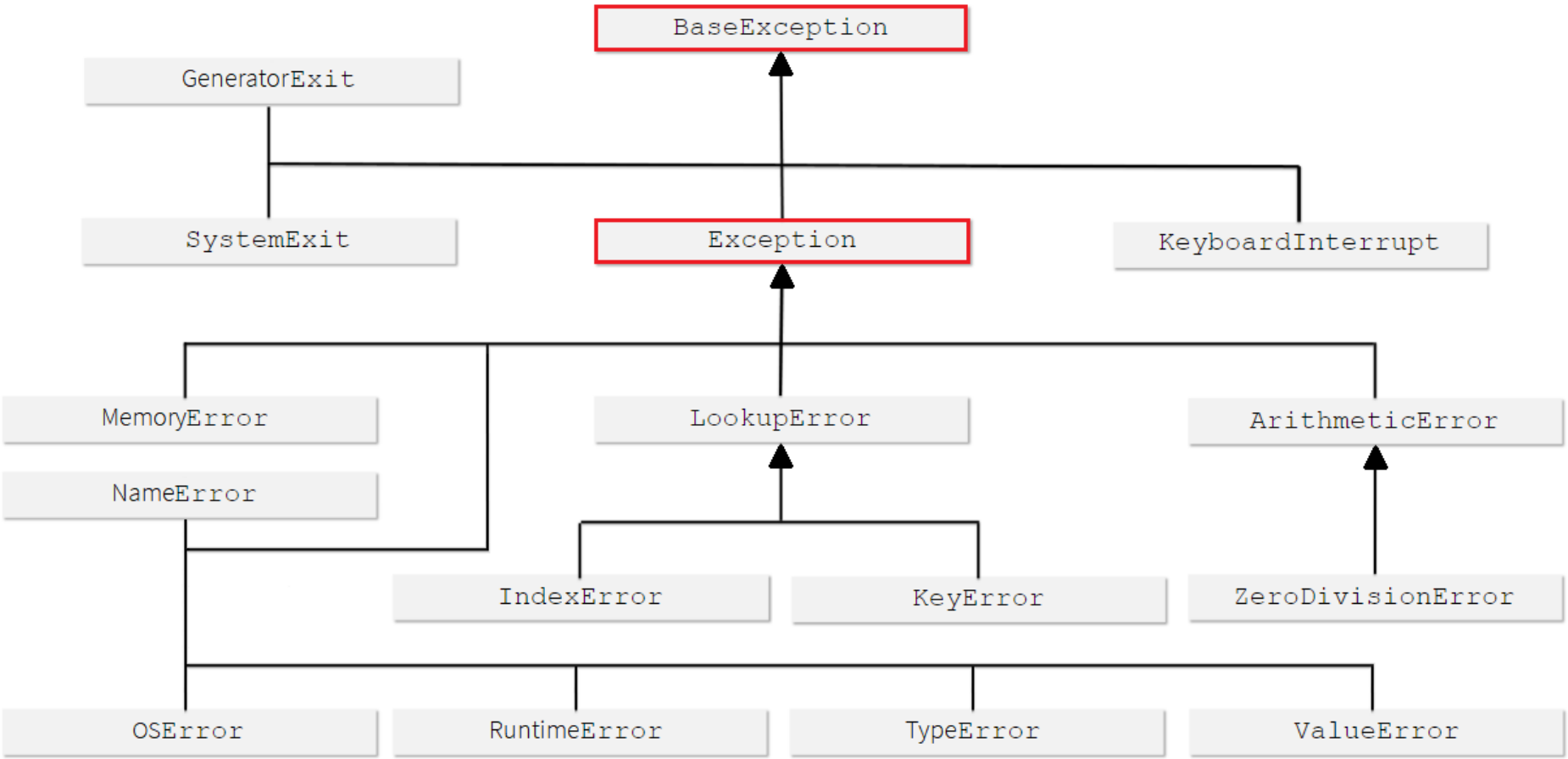
```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
```

[https://docs.python.org/3/library/exceptions.html]

Python Exceptions

- In Python, all exceptions must be instances of a class that derives from `BaseException`.
- The built-in exceptions can be generated by the interpreter or built-in functions.
- Almost all of the exceptions have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information. The associated value is usually passed as arguments to the exception class's constructor.
- User code can also raise the built-in exceptions.
- The built-in exception classes can be subclassed to define new exceptions. Programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, but not from `BaseException`.

Selected Built-in Exceptions



Selected Built-in Exceptions – `BaseException`

`BaseException`

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments (also see the `args` attribute).

Selected Built-in Exceptions – `SystemExit`

`SystemExit` $\rightarrow$ `BaseException`

This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits. No stack traceback is printed. The constructor accepts the same optional argument passed to `sys.exit()`. If the value is an integer, it specifies the system exit status (passed to C's `exit()` function). If it is `None`, the exit status is zero.

Selected Built-in Exceptions – `KeyboardInterrupt`

`KeyboardInterrupt` $\rightarrow$ `BaseException`

Raised when the user hits the interrupt key (normally `Ctrl+C` or `Delete`).

Selected Built-in Exceptions – `GeneratorExit`

`GeneratorExit` → `BaseException`

Raised when a generator or coroutine is closed. See `generator.close()` and `coroutine.close()`. It directly inherits from `BaseException` instead of `Exception` since it is technically not an error.

Selected Built-in Exceptions – `Exception`

`Exception` $\rightarrow$ `BaseException`

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

Selected Built-in Exceptions – `ArithmeticError`

`ArithmeticError` → `Exception` → `BaseException`

The base class for those built-in exceptions that are raised for various arithmetic errors:

- `ZeroDivisionError`,
- `OverflowError`,
- `FloatingPointError`.

Selected Built-in Exceptions – `ZeroDivisionError`

`ZeroDivisionError` → `ArithmeticError` → `...` → `BaseException`

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

Selected Built-in Exceptions – `OverflowError`

`OverflowError` → `ArithmeticError` → `...` → `BaseException`

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise the `MemoryError` than give up). For historical reasons, the `OverflowError` is sometimes raised for integers that are outside a required range.

Selected Built-in Exceptions – `FloatingPointError`

`FloatingPointError` → `ArithmeticError` → `...` → `BaseException`

Currently not used. Because of the lack of standardization of floating point exception handling in C, most floating point operations are not checked.

Selected Built-in Exceptions – `LookupError`

`LookupError` → `Exception` → `BaseException`

The base class for the exceptions that are raised when an index or a key used on a sequence or mapping is invalid:

- `IndexError`,
- `KeyError`.

Selected Built-in Exceptions – `IndexError`

`IndexError` → `LookupError` → `...` → `BaseException`

Raised when a sequence subscript is out of range.

Note: slice indices are silently truncated to fall in the allowed range. If an index is not an integer, `TypeError` is raised.

Selected Built-in Exceptions – `KeyError`

`KeyError` → `LookupError` → `...` → `BaseException`

Raised when a mapping (dictionary) key is not found in the set of existing keys.

Selected Built-in Exceptions – `MemoryError`

`MemoryError` → `Exception` → `BaseException`

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory.

Note: because of the underlying memory management architecture (C's `malloc()` function), the interpreter may not always be able to completely recover from this situation.

Selected Built-in Exceptions – `NameError`

`NameError` → `Exception` → `BaseException`

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

Selected Built-in Exceptions – `OSError`

`OSError` $\rightarrow$ `Exception` $\rightarrow$ `BaseException`

This exception is raised when a system function returns a system-related error, including `I/O` failures such as "file not found" or "disk full".

Selected Built-in Exceptions – `RuntimeError`

`RuntimeError` → `Exception` → `BaseException`

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

Selected Built-in Exceptions – `TypeError`

`TypeError` → `Exception` → `BaseException`

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch. This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be.

Note: if an object is meant to support a given operation but has not yet provided an implementation, `NotImplementedError` is the proper exception to raise.

Selected Built-in Exceptions – `ValueError`

`ValueError` $\rightarrow$ `Exception` $\rightarrow$ `BaseException`

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

The `try...except` Statement

```
try:
    suite
(except[ expression[ as identifier]]:
    suite)+
[else:
    suite]
[finally:
    suite]
```

- Exception handlers are specified with the `try...except` statement.
- The `finally` clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

The try...except Statement – Example

```
1  try:
2      x = int(input("Enter a divisor: "))
3      y = 1 / x
4      print("Result:", y)
5
6  except ValueError:
7      print("You have to enter an integer value!")
8  except ZeroDivisionError:
9      print("Cannot divide by zero!")
10 except:
11     print("Unexpected error!")
```

How Python Exceptions are Handled?

- First, the `try suite` is executed.
- The `except` clause(s) specify one or more exception handlers.
- When no exception occurs in the `try` clause, no exception handler is executed. The optional `else` and `finally` clauses are executed (more on that later) and execution of the `try` statement is finished.
- When an exception occurs during execution of the `try suite`, the rest of the `try suite` is skipped and a search for an exception handler is started.
- This search inspects the `except` clauses in turn until one is found that matches the exception – only the first matching `except` clause is triggered.

How Python Exceptions are Handled? – Continued

- Exceptions are identified by class instances. The `except` clause is selected depending on the class of the exception object.
- For an `except` clause with an `expression`, that `expression` is evaluated, and the clause matches the exception if the resulting object is "compatible" with the exception:
  - it must reference the class or a base class of the exception object,
  - or a tuple containing an item compatible with the exception.
- An expression-less `except` clause, if present, must be the last – it matches any exception.
- At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding `try` clause, not in other handlers of the same `try` statement.

How Python Exceptions are Handled? – Continued

- When a matching `except` clause is found, the exception is assigned to the target specified after the `as` keyword in that `except` clause, if present, and the `except` clause's `suite` is executed. When it is finished, execution continues normally after the entire `try` statement.
- When an exception has been assigned using `as` keyword, it is cleared at the end of the `except` clause.
- This means the exception must be assigned to a different name to be able to refer to it after the `except` clause.
- Before an `except suite` is executed, details about the exception are stored in the `sys` module and can be accessed via `sys.exc_info()`.
- `sys.exc_info()` returns a 3-tuple consisting of the exception class, the exception instance and a traceback object identifying the point in the program where the exception occurred.

Printing an Exception – The sys.exc_info() Function

```
1 import sys
2
3 def print_exc_info():
4     info = sys.exc_info() # It's a 3-tuple.
5     print("Exception class:", info[0])
6     print("Exception name:", info[0].__name__)
7     print("Exception instance:", info[1])
8     print("Traceback:", info[2])
9     print("Error in line:", info[2].tb_lineno)
```

Traceback object's read-only attributes

```
['tb_frame', 'tb_lasti', 'tb_lineno', 'tb_next']
```

Note: you can extract more information from a traceback object using functions defined in the traceback module.

Multiple Exceptions in a Single except Clause

```
try:
    x = int(input("Enter a divisor: "))
    y = 1 / x
    print("Result:", y)

except (ValueError, ZeroDivisionError):
    print_exc_info()
except:
    print("Unexpected error!")
    print_exc_info()
```

Multiple Exceptions in a Single except Clause – Output

Output (user input: 0)

```
Exception class: <class 'ZeroDivisionError'>
Exception name: ZeroDivisionError
Exception instance: division by zero
Traceback: <traceback object at 0x000001D55A42ED08>
Error in line: 3
```

Output (user input: Ctrl+C)

```
Unexpected error!
Exception class: <class 'KeyboardInterrupt'>
Exception name: KeyboardInterrupt
Exception instance:
Traceback: <traceback object at 0x00000129CABBDD88>
Error in line: 2
```

Reading an Integer Number Safely

```
1 while True:
2     try:
3         x = int(input("Please enter a number: "))
4         print("The number you have entered:", x)
5         break
6     except ValueError:
7         print("That was no valid number. Try again...")
```

Output (user input: "text", 100)

```
Please enter a number: text
That was no valid number. Try again...
Please enter a number: 100
The number you have entered: 100
```

Exceptions Raised in the except Clause

```
1  try:
2      x = int(input("Enter a divisor: "))
3      y = 1 / x
4      print("Result:", y)
5
6  except ValueError:
7      print("You have to enter an integer value!")
8      print(1 / 0) # Will it be handled?
9  except ZeroDivisionError:
10     print("Cannot divide by zero!")
11     print(1 / 0) # Will it be handled?
12 except:
13     print("Unexpected error!")
```

Exceptions Raised in the except Clause – Output

Output (user input: "text")

```
You have to enter an integer value!
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    x = int(input("Enter a divisor: "))
ValueError: invalid literal for int() with base 10: 'text'

During handling of the above exception,
another exception occurred:

Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print(1 / 0) # Will it be handled?
ZeroDivisionError: division by zero
```

The except Clause – The as Keyword

```
1  try:
2      x = int(input("Enter a divisor: "))
3      y = 1 / x
4      print("Result:", y)
5
6  except ValueError as e:
7      print("You have to enter an integer value!")
8      exc = e
9  except ZeroDivisionError as e:
10     print("Cannot divide by zero!")
11     exc = e
12 except:
13     print("Unexpected error!")
14
15 try:
16     print(exc)
17 except NameError as e:
18     print("There was no exception,", e)
```

The except Clause – The as Keyword – Output

Output (user input: "text")

```
Enter a divisor: text
You have to enter an integer value!
invalid literal for int() with base 10: 'text'
```

Output (user input: "0")

```
Enter a divisor: 0
Cannot divide by zero!
division by zero
```

Output (user input: Ctrl+C)

```
Enter a divisor:
Unexpected error!
There was no exception, name 'exc' is not defined
```

How Python Exceptions are Handled? – Continued

- If no `except` clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. If no handler is found, it is an *unhandled exception* and execution stops.
- The optional `else` clause is executed if the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue` or `break` statement was executed. Exceptions raised in the `else` clause are not handled by the preceding `except` clauses.
- If an exception occurs in any of the `try`, `except` and `else` clauses and is not handled, the exception is temporarily saved, and then the `finally` clause, if present, is executed. If there is a saved exception it is re-raised at the end of the `finally` clause.
- If the `finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `finally` clause executes a `return` or `break` statement, the saved exception is discarded.
- A `continue` statement is illegal in the `finally` clause.

Python Interpreter Error Handling

- Python uses the "termination" model of error handling:
  - an exception handler can find out what happened and continue execution at an outer level,
  - but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).
- When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop.
- In either case, it prints a stack traceback, except when the exception is `SystemExit`.
- If the evaluation of an `expression` in the header of an `except` clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

The else and finally Clauses

```
1  try:
2      x = int(input("Enter a divisor: "))
3      y = 1 / x
4      print("Result:", y)
5
6  except ValueError:
7      print("You have to enter an integer value!")
8  except ZeroDivisionError:
9      print("Cannot divide by zero!")
10 else:
11     print("I'm in the else branch.")
12 finally:
13     print("Finally, I'm here!")
14
15 print("THE END")
```

The else and finally Clauses – Output

Output (user input: "text")

```
You have to enter an integer value!
Finally, I'm here!
THE END
```

Output (user input: 0)

```
Cannot divide by zero!
Finally, I'm here!
THE END
```

The else and finally Clauses – Output

Output (user input: 10)

```
Result: 0.1
I'm in the else branch.
Finally, I'm here!
THE END
```

Output (user input: Ctrl+C)

```
Finally, I'm here!
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    x = int(input("Enter a divisor: "))
KeyboardInterrupt
```

Handling Exceptions Raised Inside Functions

```
1 def func():
2     x = int(input("Enter a divisor: "))
3     y = 1 / x
4     print("Result:", y)
5
6 try:
7     func()
8 except Exception as e:
9     print(e)
```

Output (user input: 0)

```
division by zero
```

Output (user input: "text")

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

The `try...finally` Statement

```
try:
    suite
finally:
    suite
```

- When a `return`, `break` or `continue` statement is executed in the `try` `suite`, the `finally` clause is also executed 'on the way out'.
- The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement in the `finally` clause will always be the last one executed.

Note: the exception information is not available to the program during execution of the `finally` clause, i.e. the `sys.exc_info()` function returns `(None, None, None)`.

The try...finally Statement – Example

```
1 def func():
2     try:
3         return "try"
4     finally:
5         return "finally"
6
7 print(func())
```

Output

```
finally
```

The try...finally Statement – Another Example

```
1 def func():
2     try:
3         return 1 / 0
4     finally:
5         return 100 # Saved exception will be discarded.
6
7 print(func())
```

Output

100

The raise Statement

```
raise[ expression[ from expression]]
```

- If no expressions are present, raise re-raises the last exception that was active in the current scope.
- If no exception is active in the current scope, a RuntimeError exception is raised indicating that this is an error.
- Otherwise, raise evaluates the first expression as the exception object. It must be either a subclass or an instance of BaseException.
- A traceback object is normally created automatically when an exception is raised and attached to it as the __traceback__ attribute, which is writable.
- You can create an exception and set your own traceback in one step using the with_traceback() exception method.

The raise Statement – Example

```
1 >>> raise NameError("Hello!")
```

Output

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise NameError("Hello!")
NameError: Hello!
```

The raise Statement – Another Example

```
1  def func(n):
2      try:
3          res = 1 / n
4      except:
5          print("func(): except")
6          raise
7      else:
8          print("func(): else branch")
9          raise Exception
10     return res # Will it ever be executed?
11
12 try:
13     print(func(int(input("Enter a divisor: "))))
14 except ArithmeticError:
15     print("global: ArithmeticError")
16 except Exception:
17     print("global: Exception!")
18 print("THE END")
```

The raise Statement – Another Example – Output

Output (user input: 0)

```
func(): except
global: ArithmeticError
THE END
```

Output (user input: 10)

```
func(): else branch
global: Exception
THE END
```

Output (user input: "text")

```
global: Exception
THE END
```

The raise Statement – Exception Chaining

```
raise[ expression[ from expression]]
```

- The `from` clause is used for exception chaining: if given, the second `expression` must be another exception class or instance, which will then be attached to the raised exception as the `__cause__` attribute (which is writable).
- If the raised exception is not handled, both exceptions will be printed.
- A similar mechanism works implicitly if an exception is raised inside an exception handler or a `finally` clause – the previous exception is then attached as the new exception's `__context__` attribute.
- Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause.

Exception Chaining – Example

```
1 try:
2     print(1 / 0)
3 except Exception as e:
4     raise RuntimeError("Something bad happened") from e
```

Output

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "test.py", line 4, in <module>
    raise RuntimeError("Something bad happened") from e
RuntimeError: Something bad happened
```

Exception Chaining – Another Example

```
1  def func():
2      try:
3          print(1 / 0)
4      except ArithmeticError as e:
5          raise TypeError("Hello!") from e
6      finally:
7          raise ValueError("Hello again!")
8
9  try:
10     func()
11 except Exception as e:
12     raise Exception("Bye-bye!") from e
```

Exception Chaining – Another Example – Output

Output

```
Traceback (most recent call last):
  File "test.py", line 3, in func
    print(1 / 0)
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "test.py", line 5, in func
    raise TypeError("Hello!") from e
TypeError: Hello!

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "test.py", line 9, in <module>
    func()
  File "test.py", line 7, in func
    raise ValueError("Hello again!")
ValueError: Hello again!

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "test.py", line 11, in <module>
    raise Exception("Bye-bye!") from e
Exception: Bye-bye!
```

Exception Arguments

```
1 try:
2     raise Exception("spam", 1, -1.23, [0, 1, 2])
3 except Exception as e:
4     print(e.args) # Prints arguments.
5     print(e) # Prints arguments.
6     for no, arg in zip(range(len(e.args)), e.args):
7         print("Exception argument ", no, ": ", arg, sep="")
```

Output

```
('spam', 1, -1.23, [0, 1, 2])
('spam', 1, -1.23, [0, 1, 2])
Exception argument 0: spam
Exception argument 1: 1
Exception argument 2: -1.23
Exception argument 3: [0, 1, 2]
```

Digression – Exception Messages

The exception object received by the `except` clause can carry a message – an additional information about the exceptional condition.

Note: exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

Plan of the Presentation

What is an Assertion?

Assertion

An assertion is a statement in the programming language that enables you to test your assumptions about your program. Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error.

The assert Statement

```
assert expression[, expression]
```

- `assert` statements are a convenient way to insert debugging assertions into a program.
- When an `assert` statement fails, i.e. its `expression` evaluates to `False`, an `AssertionError` is raised.

The assert Statement – Simple and Extended Form

The simple form, `assert expression`, is equivalent to:

```
if __debug__:
    if not expression:
        raise AssertionError
```

The extended form, `assert expression_1, expression_2`, is equivalent to:

```
if __debug__:
    if not expression_1:
        raise AssertionError(expression_2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names.

Python Assertions

- In the current implementation, the built-in variable `__debug__` is:
  - `True` – under normal circumstances,
  - `False` – when optimization is requested (command line option `-O`).
- The current code generator emits no code for an `assert` statement when optimization is requested at compile time.
- Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.
- Note that it is unnecessary to include the source code for the expression that failed in the error message – it will be displayed as part of the stack trace.

The assert Statement – Example

```
1 import math
2
3 x = float(input("Enter floating-point number: "))
4 assert x >= 0.0
5 print("sqrt(x):", math.sqrt(x))
```

Output (user input: -1)

```
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    assert x >= 0.0
AssertionError
```

The assert Statement – Another Example

```
1  def func():
2      x = int(input("Enter a divisor: "))
3      assert x != 0, "Cannot divide by zero!"
4      y = 1 / x
5      print("Result:", y)
6
7  try:
8      func()
9  except Exception as e:
10     print("Exception:", e)
```

Output (user input: 0)

```
Enter a divisor: 0
Exception: Cannot divide by zero!
```

Plan of the Presentation

The str Type

- Textual data in Python 3 is handled with `str` objects, or *strings*.
- Strings are *immutable sequences of Unicode code points*.
- Code points must be below 1114112 (which is the full Unicode range).
- Since there is no separate "character" type, indexing a string produces strings of length 1.
- Unicode encoded characters can be used to name variables, functions and other entities.
- They can also be used during all input and output operations.
- Strings can be compared using the same set of operators which are in use in relation to numbers (`<`, `<=`, `>`, `>=`, `!=`, `==`).

Note: the character set of a source code is defined by the encoding declaration. It is `UTF-8` if no encoding declaration is given in the source file. See section in the Python documentation regarding encoding declarations.

Glossary of Unicode Terms

- *Coded Character Set* – a character set in which each character is assigned a numeric code point. Frequently abbreviated as character set, charset, or code set. The acronym CCS is also used.
- *Code Page* – a coded character set, often referring to a coded character set used by a personal computer.
- *Codespace* – a range of numerical values available for encoding characters. For the Unicode Standard, a range of integers from 0 to $10FFFF_{16}$.
- *Code Point* – any value in the Unicode codespace. More generally, a value, or position, for a character, in any coded character set.
- `http://unicode.org/glossary/`

String Type and Identity

```
1  str_1 = "Monty Python!"
2  str_2 = "Monty Python!"
3
4  print(type(str_1), type(str_2))
5  print(id(str_1))
6  print(id(str_2))
7  print(str_1 is str_2)
8
9  str_1 = ""
10 print(str_1 is str_2)
```

Possible output

```
<class 'str'> <class 'str'>
1818158982960
1818158982960
True
False
```

String Literals

```
1 print('Single quotes allow embedded "double" quotes.')
2 print("Double quotes allow embedded 'single' quotes.")
3 print('''Three single quotes.''')
4 print("""Three
5     "double""\
6 quotes.""")
```

Output

```
Single quotes allow embedded "double" quotes.
Double quotes allow embedded 'single' quotes.
Three single quotes.
Three
    "double""quotes.
```

String Literals – Implicit Concatenation

```
1 if ("one " 'two' == "one two"):
2     print("String literals are equal.")
3 else:
4     print("String literals are not equal.")
```

Output

```
String literals are equal.
```

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal.

String Literals – Implicit Concatenation – Another Example

```
1  string_1 = ("AAA"  # First part of a string.
2              "BBB"  # Second part of a string.
3              "CCC") # Third part of a string.
4
5  string_2 =  "CCC"\
6              "BBB"\
7              "AAA"
8
9  print(string_1)
10 print(string_2)
```

Output

```
AAABBBCCC
CCCBBBAAA
```

Raw Strings

```
1 str_1 = r"First line \n Second line"
2 str_2 = r"Double quotes escaped \""
3
4 print(str_1)
5 print(str_2)
```

Output

```
First line \n Second line
Double quotes escaped \"
```

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result.

String's Length – The `len()` Function

```
1 str_1 = "Double quotes."
2 str_2 = """Three
3     "double""
4 quotes."""
5
6 print(len(str_1))
7 print(len(str_2))
```

Output

```
14
24
```

String Overloaded Operators

Operator	Operation	Usage
+	concatenation	`string_1 + string_2`
*	replication	`string * integer_number`
		`integer_number * string`

String Concatenation

```
1 fn = input("May I have your first name, please?\n")
2 ln = input("May I have your last name, please?\n")
3
4 message = "Your name is "
5 message += ln
6 message += ". " + fn
7 message += " " + ln + "."
8
9 print(message)
```

Output (user input: "James", "Bond")

```
Your name is Bond. James Bond.
```

To efficiently construct strings from multiple fragments, `str.join()` or `io.StringIO` can be used.

The `join()` Method

`S.join(iterable)`

- Returns a string which is the concatenation of the strings in `iterable`.
- The separator between elements is the string providing this method.

The join() Method – Example

```
1 fn = input("May I have your first name, please?\n")
2 ln = input("May I have your last name, please?\n")
3
4 strings = ("Your name is ", ln, ". ", fn, " ", ln, ".")
5 message_1 = str("").join(strings) # String constructor.
6 message_2 = "#".join(strings) # Will it work?
7
8 print(message_1)
9 print(message_2)
```

Output (user input: "James", "Bond")

```
Your name is Bond. James Bond.
Your name is #Bond#. #James# #Bond#.
```

String Replication

```
1 print('+' + 50 * '-' + '+')
2 print(('|' + ' ' * 50 + '|\n') * 5, end='')
3 print('+' + 50 * '-' + '+')
```

Output

```
+--------------------------------------------------+
|                                                  |
|                                                  |
|                                                  |
|                                                  |
|                                                  |
+--------------------------------------------------+
```

The `ord()` Function

`ord(c)`

- Returns the Unicode code point for a one-character string `c`.
- When a multi-character string is passed as an argument, the `TypeError` is raised.

The ord() Function – Example

```
1 c_1 = 'B'
2 c_2 = 'β'
3
4 print(ord(c_1))
5 print(ord(c_2))
```

Output

66

946

The chr() Function

`chr(i)`

Returns a Unicode string of one character with a code point `i`.

The chr() Function – Example

```
1 print(chr(66))
2 print(chr(946))
```

Output

B

β

How to Use a String?

```
1  s = "aaAbBCCCDaaEfghhaaH"
2
3  print("s[0]:", s[0], end=", ")
4  print("s[-2]:", s[-2], end=", ")
5  print("s[5:8]:", s[5:8])
6  print("len(s):", len(s), end=", ")
7  print("min(s): ", min(s), ", max(s):", max(s), sep="")
8  print("th" in s, "ht" not in s)
9  s = "".join([chr(ord(c)-10) for c in s])
10 for c in s:
11     print(c, end="")
```

Output

```
s[0]: a, s[-2]: a, s[5:8]: CCC
len(s): 19, min(s): A, max(s):h
False True
WW7X8999:WW;\]^^WW>
```

String – Unsupported Operations

```
1 s = "aaAbBCCCDaaEfghhaaH"
2
3 s[-1] = 0 # Error!
4 del s[0] # Error!
5 s[0], s[-1] = s[-1], s[0] # Error!
```

Output

```
TypeError: 'str' object does not support item assignment
(lines: 3, 5)
TypeError: 'str' object doesn't support item deletion
(line 4)
```

List of <class 'str'> Methods

capitalize()	center()	count()
encode()	endswith()	find()
format()	index()	is*()
join()	lower()	partition()
replace()	split()	splitlines()
startswith()	strip()	swapcase()
title()	translate()	upper()
zfill()	A few more...	

String vs. List – Unsupported Methods

- `append()`
- `clear()`
- `copy()`
- `extend()`
- `insert()`
- `pop()`
- `remove()`
- `reverse()`
- `sort()`

The capitalize() Method

```
S.capitalize() -> str
```

- Returns a capitalized version of the string.
- More specifically, makes the first character have upper case and the rest lower case.

The capitalize() Method – Example

```
1 s_1 = "test string"
2 s_2 = "TESTSTRING"
3 s_3 = " teststring"
4 s_4 = "4815162342"
5
6 print(s_1.capitalize())
7 print(s_2.capitalize())
8 print(s_3.capitalize())
9 print(s_4.capitalize())
```

Output

```
Test string
Teststring
 teststring
4815162342
```

The center() and zfill() Methods

```
S.center(width, fillchar=' ') -> str
S.zfill(width) -> str
```

- center() returns a centered string of length width:
  - padding is done using the specified fill character (default is a space),
  - the original string is returned if width is less than or equal to len(S).
- zfill() pads a numeric string with zeros on the left, to fill a field of the given width. The string is never truncated.

The center() and zfill() Methods – Example

```
1 s = "Test string"
2 print("[" + s.center(20) + "]")
3 print("[" + s.center(25, '#') + "]")
4 print("[" + s.center(5, '@') + "]")
5 print("23".zfill(5))
6 print("-23".zfill(5))
7 print("4815162342".zfill(5))
```

Output

```
[    Test string     ]
[#######Test string#######]
[Test string]
00023
-0023
4815162342
```

The count() and index() Methods

```
S.count(sub[, start[, end]]) -> int
S.index(sub[, start[, end]]) -> int
```

- `count()` returns the number of non-overlapping occurrences of substring `sub` in string `S[start:end]`.
- `index()` returns the lowest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Raises `ValueError` when the substring is not found.
- Optional arguments `start` and `end` are interpreted as in slice notation.
- See also the `rindex()` method.

The count() and index() Methods – Example

```
1 s = "aaAbBCCCDaaEfghhaaH"
2
3 print("count:", [(c, s.count(c)) for c in ('a', 'C')])
4 print("count('CC'):", s.count('CC'))
5 print("count('aa', 0, 11):", s.count('aa', 0, 11))
6 print("index('C'):", s.index('C'))
7 print("index('aaE'):", s.index('aaE'))
```

Output

```
count: [('a', 6), ('C', 3)]
count('CC'): 1
count('aa', 0, 11): 2
index('C'): 5
index('aaE'): 9
```

The encode() Method

```
S.encode(encoding='utf-8', errors='strict') -> bytes
```

- Encodes the string using the codec registered for `encoding`.
- Returns a `bytes` object.
- `encoding` – the encoding in which to encode the string.
- `errors` – the error handling scheme to use for encoding errors. The default is `'strict'` – encoding errors raise a `UnicodeEncodeError`.
- Other possible values of `errors`:
  - `'ignore'`,
  - `'replace'`,
  - `'xmlcharrefreplace'`,
  - `'backslashreplace'`,
  - as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.
- For a list of possible encodings, see section about standard encodings in the Python documentation.

The find() Method

```
S.find(sub[, start[, end]]) -> int
```

- Returns the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
- Optional arguments start and end are interpreted as in slice notation.
- Returns -1 on failure.
- See also the rfind() method.

The find() Method – Example

```
1  s = ("Lorem ipsum dolor sit amet, consectetur adipiscing "
2       "elit, sed do eiusmod tempor incididunt ut labore et "
3       "dolore magna aliqua. Ut enim ad minim veniam, quis "
4       "nostrud exercitation ullamco laboris nisi ut aliquip "
5       "ex ea commodo consequat. Duis aute irure dolor in "
6       "reprehenderit in voluptate velit esse cillum dolore "
7       "eu fugiat nulla pariatur.")
8
9  str_to_find = "in"
10 idx = s.find(str_to_find)
11 while idx != -1:
12     print(idx, end=" ")
13     idx = s.find(str_to_find, idx+1)
```

Output

```
47 79 136 254 271
```

The format() Method

```
S.format(*args, **kwargs) -> str
```

- Returns a formatted version of `S`, using substitutions from `args` and `kwargs`.
- The substitutions are identified by replacement fields delimited by braces `{}`.
- Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument.

The format() Method – Example

```
1 s_1 = "The sum of {0} + {1} is {2}."
2 s_2 = "The multiplication of {n1} by {n2} is {res}."
3
4 print(s_1.format(2, 3, 2+3))
5 print(s_1.format(20, 30, 20+30))
6 print(s_2.format(n1=4, n2=5, res=4*5))
7 print(s_2.format(n2=10, n1=2, res=2*10))
```

Output

```
The sum of 2 + 3 is 5.
The sum of 20 + 30 is 50.
The multiplication of 4 by 5 is 20.
The multiplication of 2 by 10 is 20.
```

Digression – Literal String Interpolation

`f"[text]{expression[!s|!r|!a][:format_specifier]}[text]..."`

- It is a new string formatting mechanism available since Python 3.6.
- Such strings are referred to as *f-strings*, taken from the leading character used to denote such strings, and standing for "formatted strings".
- F-strings provide a concise, readable way to embed the value of Python expressions inside strings.
- The expressions are placed within strings inside braces.
- After each expression, an optional type conversion may be specified:
  - `!s` – it calls `str()`,
  - `!r` – it calls `repr()`,
  - `!a` – it calls `ascii()`.

Note: f-string is really an expression evaluated at run time, not a literal.

Digression – Literal String Interpolation – Example

```
1  val = 12.34567
2  s = "AAAaaaAaa"
3  l = 'a'
4  width = 10
5  precision = 4
6
7  print(f"The sum of {1} + {val} is {1+val}.")
8  print(f"Letter '{l}' occures {s.count(l)}x in '{s}'.")
9  s = f"result: {val:{width}.{precision}}"
10 print(s)
```

Output

```
The sum of 1 + 12.34567 is 13.34567.
Letter 'a' occures 5x in 'AAAaaaAaa'.
result:      12.35
```

The is*() Methods

Method	Description
isalnum()	Returns True if all characters in the string are alphanumeric and there is at least one character, False otherwise. A character c is alphanumeric if one of the following returns True: c.isalpha(), c.isdecimal(), c.isdigit(), or c.isnumeric().
isalpha()	Returns True if all characters in the string are alphabetic and there is at least one character, False otherwise.
isascii()	Returns True if the string is empty or all characters in the string are ASCII, False otherwise. ASCII characters have code points in the range U+0000–U+007F.
isdecimal()	Returns True if all characters in the string are decimal characters and there is at least one character, False otherwise. Decimal characters are those that can be used to form numbers in base 10.

The is*() Methods – Continued

Method	Description
`isdigit()`	Returns `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Digits include decimal characters.
`isidentifier()`	Returns `True` if the string is a valid identifier according to the language definition. Use `keyword.iskeyword()` to test for reserved identifiers such as `def` and `class`.
`islower()`	Returns `True` if all cased characters (uppercase, lowercase and titlecase letters) in the string are lowercase and there is at least one cased character, `False` otherwise.
`isnumeric()`	Returns `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise.

The is*() Methods – Continued

Method	Description
isprintable()	Returns True if all characters in the string are printable or the string is empty, False otherwise.
isspace()	Returns True if there are only whitespace characters in the string and there is at least one character, False otherwise.
istitle()	Returns True if the string is a titlecased string and there is at least one character, False otherwise. For example, uppercase characters may only follow uncased characters and lowercase characters only cased ones.
isupper()	Returns True if all cased characters (uppercase, lowercase and titlecase letters) in the string are uppercase and there is at least one cased character, False otherwise.

The lower/upper/swapcase/title() Methods

```
S.lower() -> str
S.upper() -> str
S.swapcase() -> str
S.title() -> str
```

- `lower()` returns a copy of the string with all the cased characters converted to lowercase.
- `upper()` returns a copy of the string with all the cased characters converted to uppercase.
- `swapcase()` converts uppercase characters to lowercase and lowercase characters to uppercase.
- `title()` returns a version of the string where each word is titlecased.

The lower/upper/swapcase/title() Methods – Example

```
1 s = "Test STRING 100#"
2
3 print(s.lower())
4 print(s.upper())
5 print(s.lower().upper())
6 print(s.swapcase())
7 print(s.title())
```

Output

```
test string 100#
TEST STRING 100#
TEST STRING 100#
tEST string 100#
Test String 100#
```

The partition() Method

```
S.partition(sep) -> tuple
```

- Partitions the string into three parts using the given separator.
- This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.
- If the separator is not found, returns a 3-tuple containing the original string and two empty strings.
- See also the `rpartition()` method.

The partition() Method – Example

```
1 s = "Test string 100#"
2
3 print(s.partition("string"))
4 print(s.partition("abc"))
```

Output

```
('Test ', 'string', ' 100#')
('Test string 100#', '', '')
```

The replace() Method

```
S.replace(old, new, count=-1) -> str
```

- Returns a copy with all occurrences of substring `old` replaced by `new`.
- If the optional argument `count` is given, only the first `count` occurrences are replaced, otherwise all.

The replace() Method – Example

```
1 s = "abracadabra abracadabra"
2
3 print(s.replace("ab", "AB"))
4 print(s.replace("ra", "RA", 3))
5 print(s.replace("abc", "ABC"))
```

Output

```
ABracadABra ABracadABra
abRAcadabRA abRAcadabra
abracadabra abracadabra
```

The split() and splitlines() Methods

```
S.split(sep=None, maxsplit=-1) -> list
S.splitlines(keepends=False) -> list
```

- split() returns a list of the words in the string, using sep as the delimiter string:
  - sep – the delimiter according which to split the string; None (the default value) means split according to any whitespace, and discard empty strings from the result,
  - if maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements), otherwise all possible splits are made,
  - see also the rsplit() method.
- splitlines() returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and is True.

The split() and splitlines() Methods – Example

```
1 print("1 2 3".split(), "1    2    3".split(), "".split())
2 print("1,2,3".split(','), "1,2,3".split(',', 1))
3 print("1,,2,,,3".split(','))
4 print('ab c\n\nde fg\rkl\r\n'.splitlines())
5 print('ab c\n\nde fg\rkl\r\n'.splitlines(True))
```

Output

```
['1', '2', '3'] ['1', '2', '3'] []
['1', '2', '3'] ['1', '2,3']
['1', '', '2', '', '', '3']
['ab c', '', 'de fg', 'kl']
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

The startswith() and endswith() Methods

```
S.startswith(prefix[, start[, end]]) -> bool
S.endswith(suffix[, start[, end]]) -> bool
```

- `startswith()` returns `True` if `S` starts with the specified `prefix`, `False` otherwise.
- `endswith()` returns `True` if `S` ends with the specified `suffix`, `False` otherwise.
- With optional `start`, they test `S` beginning at that position.
- With optional `end`, they stop comparing `S` at that position.
- `prefix` and `suffix` can also be a tuple of strings to try.

The startswith() and endswith() Methods – Example

```
1 list = ["draw_line()", "draw_circle()",
2         "fill_circle()", "fill_square()",
3         "line", "arc", "circle", "triangle"]
4 print("1:",[s for s in list if s.startswith("draw")])
5 print("2:",[s for s in list if s.startswith("circle", -8)])
6 print("3:",[s for s in list if s.endswith("()")])
7 print("4:",[s for s in list if s.endswith("rc", 0, 4)])
8 print("5:",[s for s in list if s.endswith(("ne", "le"))])
```

Output

```
1: ['draw_line()', 'draw_circle()']
2: ['draw_circle()', 'fill_circle()', 'circle']
3: ['draw_line()', 'draw_circle()',
    'fill_circle()', 'fill_square()']
4: ['arc', 'circle']
5: ['line', 'circle', 'triangle']
```

The strip() Method

```
S.strip(chars=None)
```

- Returns a copy of the string with leading and trailing whitespace removed.
- If `chars` is given and not `None`, it removes characters in `chars` instead.
- See also the `lstrip()` and `rstrip()` methods.

The strip() Method – Example

```
1 print("     spacious     ".strip())
2 print("www.example.com".strip("cmowz."))
```

Output

```
spacious
example
```

The translate() Method

```
S.translate(table) -> str
```

- Replaces each character in the string using the given translation `table`.
- Translation `table` must be a mapping of Unicode ordinals to Unicode ordinals, strings, or `None`.
- The `table` must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched.
- Characters mapped to `None` are deleted.

The translate() Method – Example

```
1 s = "This string WILL BE translated."
2 table = {ord('B'): 'be', ord('E'): 'en', ord('I'): 'as',
3           ord('L'): None, ord('W'): 'h'}
4
5 print(s)
6 print(s.translate(table))
```

Output

```
This string WILL BE translated.
This string has been translated.
```

Plan of the Presentation

Questions

?

The End

Thank you for your attention.