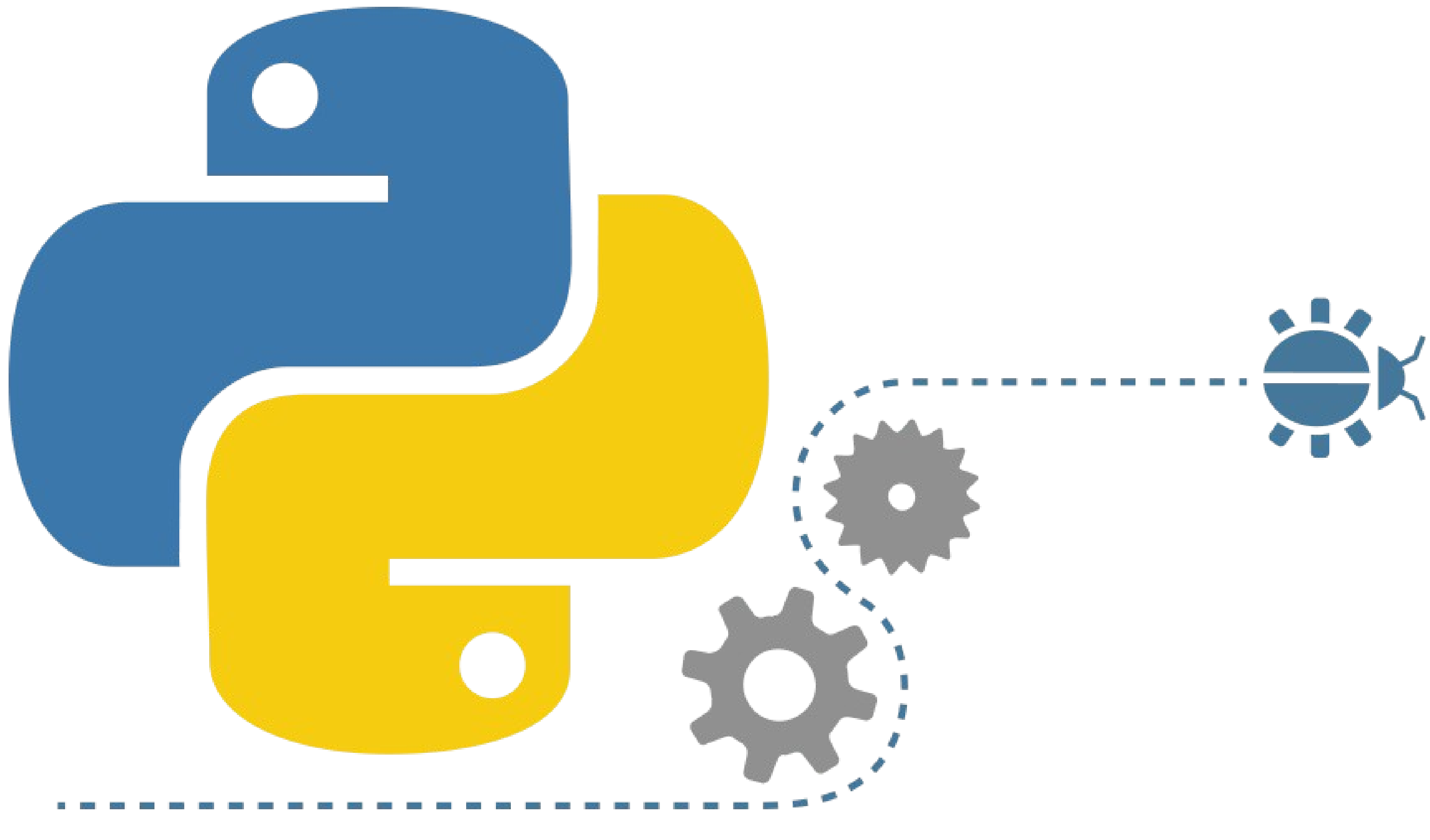

Excepciones

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

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

>>> f = open("archivo.txt","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'archivo.txt'
```

Excepciones

```
>>> 1/0

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    1/0

ZeroDivisionError: integer division or modulo by zero
```

- **Exception:** An error that occurs during the execution of a program
- Exception is **raised** and can be **caught** (or **trapped**) then **handled.**
- Unhandled, halts program and error message displayed

try / except Clause

```
try:
    num = float(input("Enter a number: "))
except:
    print("Something went wrong!")
```

Exception Type

```
try:
    num = float(input("\nEnter a number: "))
except(ValueError):
    print("That was not a number!")
```

Different types of errors raise different types of exceptions

`except` clause can specify exception types to handle

Attempt to convert `"Hi!"` to float raises `ValueError` exception

Good programming practice to specify exception types to handle each individual case

Avoid general, catch-all exception handling

Selected Exception Types

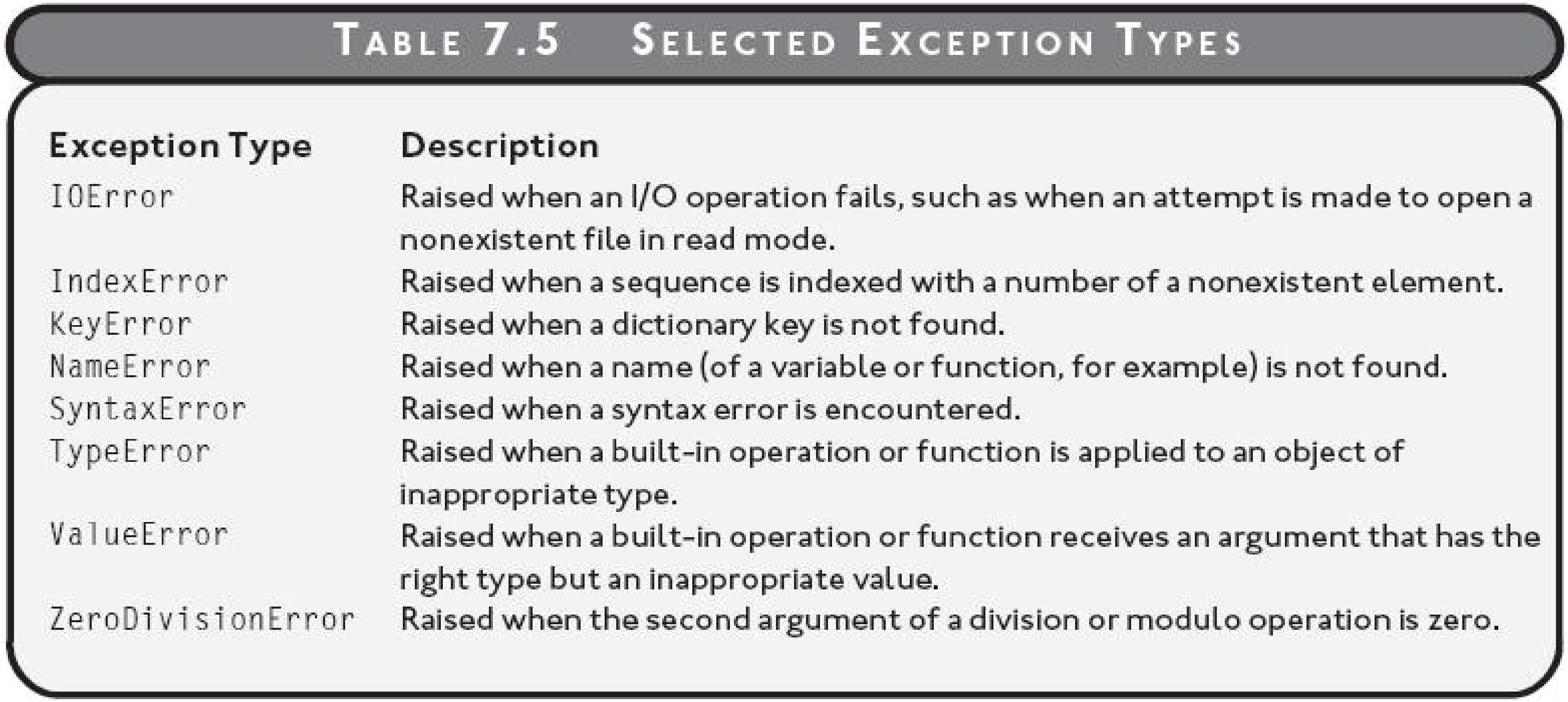

TABLE 7.5 SELECTED EXCEPTION TYPES

Exception Type	Description
`IOError`	Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode.
`IndexError`	Raised when a sequence is indexed with a number of a nonexistent element.
`KeyError`	Raised when a dictionary key is not found.
`NameError`	Raised when a name (of a variable or function, for example) is not found.
`SyntaxError`	Raised when a syntax error is encountered.
`TypeError`	Raised when a built-in operation or function is applied to an object of inappropriate type.
`ValueError`	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero.

Exception Types

```
for value in (None, "Hi!", 5):
    try:
       print("Attempting to convert {} –>{}".
                       format(value, float(value)))
    except(TypeError, ValueError):
        print "Something went wrong!"
```

Can trap for multiple exception types

Can list different exception types in a single `except` clause

Code will catch either `TypeError` or `ValueError` exceptions

Exception Types

```
for value in (None, "Hi!", 5):
   try:
       print("Attempting to convert {} –>{}".
                      format(value, float(value)))
   except(TypeError):
       print("Can only convert string or number!")
   except(ValueError):
       print("Can only convert a string of digits!")
```

Another method to trap for multiple exception types is multiple `except` clauses after single `try`

Each `except` clause can offer specific code for each individual exception type

Getting Exception info

```
try:
  num = float(input("\nEnter a number: "))
except ValueError as e:
  print("Not a number! Or as Python would say: {}"
                                            .format(e))
```

Exception may have an argument, usually message describing exception

Get the argument if a variable is listed before the colon in `except` statement

Adding an `else` Clause

```
try:
    num = float(input("\nEnter a number: "))
except(ValueError):
    print("That was not a number!")
else:
    print("You entered the number {}".format(num))
```

Can add single `else` clause after all `except` clauses

`else` block executes only if no exception is raised

`num` printed only if assignment statement in the `try` block raises no exception

Passing arguments

- The Python sys module provides access to any command-line arguments via the sys.argv
  - sys.argv is the list of command-line arguments.
  - len(sys.argv) is the number of command-line arguments.
  - sys.argv[0] is the program ie. the script name.

Passing arguments

```
import sys


print ('Number of arguments:', len(sys.argv), 'arguments.')
print ('Argument List:', str(sys.argv))
```

getopt

- Python provides a getopt module that helps you parse command-line options and arguments.

getopt.getopt(args, options, [long_options])

- **args** − This is the argument list to be parsed.
- **options** − This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options** − This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

getopt

```
opts, args = getopt.getopt(sys.argv[1:],"hi:o:",["ifile=","ofile="])
for opt, arg in opts:
    if opt == '-h':
        print ('test.py -i <inputfile> -o <outputfile>')
        sys.exit()
    elif opt in ("-i", "--ifile"):
        inputfile = arg
    elif opt in ("-o", "--ofile"):
        outputfile = arg
```

__main__

```
def main()
    print(‘Hello world!’)

if __name__ == "__main__":
    main(sys.argv[1:])
```

```
import sys, getopt

def main(argv):
    inputfile = ''
    outputfile = ''
    try:
        opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
    except getopt.GetoptError:
        print ('copy.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print ('test.py -i <inputfile> -o <outputfile>')
            sys.exit()
        elif opt in ("-i", "--ifile"):
            inputfile = arg
        elif opt in ("-o", "--ofile"):
            outputfile = arg
    print ('Input file is {}’.format(inputfile))
    print ('Output file is {}‘.format(outputfile))

if __name__ == "__main__":
    main(sys.argv[1:])
```