

# Exceptions and file input/output

- try-raise-except-finally
- Exception
- control flow
- match - case
- file open/read/write
- sys.stdin, sys.stdout, sys.stderr
- context manager

# Exceptions – Error handling and control flow

- **Exceptions** is a widespread technique to handle run-time **errors** / abnormal behaviour (e.g. in Python, Java, C++, JavaScript, C#)
- **Exceptions** can also be used as an **advanced control flow mechanism** (e.g. in Python, Java, JavaScript)
  - *Problem: How to perform a "break" in a recursive function ?*

# Built-in exceptions (class hierarchy)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- TypeError
    +-- ValueError
        |   +-- UnicodeError
        |       +-- UnicodeDecodeError
        |       +-- UnicodeEncodeError
        |       +-- UnicodeTranslateError
```

```
+-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |       |   +-- BrokenPipeError
    |       |   +-- ConnectionAbortedError
    |       |   +-- ConnectionRefusedError
    |       |   +-- ConnectionResetError
    +-- FileNotFoundError
    +-- InterruptedError
    +-- IsADirectoryError
    +-- NotADirectoryError
    +-- PermissionError
    +-- ProcessLookupError
    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
    |   +-- NotImplementedError
    |   +-- RecursionError
+-- SyntaxError
    |   +-- IndentationError
    |   +-- TabError
+-- SystemError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

# Typical built-in exceptions

and unhandled  
behaviour

Python shell

```
> 7 / 0
| ZeroDivisionError: division by zero
> int('42x')
| ValueError: invalid literal for int() with base 10: '42x'
> x = y
| NameError: name 'y' is not defined
> L = [1] * 10_000_000_000
| MemoryError
> 2.5 ** 1000
| OverflowError: (34, 'Result too large')
> t = (3, 4)
> t[0] = 7
| TypeError: 'tuple' object does not support item assignment
> t[3]
| IndexError: tuple index out of range
> t.x
| AttributeError: 'tuple' object has no attribute 'x'
> x = {}
> x['foo']
| KeyError: 'foo'
> def f(x): f(x + 1)
> f(0)
| RecursionError: maximum recursion depth exceeded
> def f(): x = x + 1
> f()
| UnboundLocalError: local variable 'x' referenced before assignment
```

# Catching exceptions – Fractions (I)

```
fraction1.py
```

```
while True:  
    numerator = int(input('Numerator = '))  
    denominator = int(input('Denominator = '))  
    result = numerator / denominator  
    print(f'{numerator} / {denominator} = {result}')
```

```
Python shell
```

```
| Numerator = 10  
| Denominator = 3  
| 10 / 3 = 3.3333333333333335  
| Numerator = 20  
| Denominator = 0  
| ZeroDivisionError: division by zero
```

# Catching exceptions – Fractions (II)

`fraction2.py`

```
while True:
    numerator = int(input('Numerator = '))
    denominator = int(input('Denominator = '))
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print(f'{numerator} / {denominator} = {result}')
```

catch  
exception

`Python shell`

```
| Numerator = 10
| Denominator = 0
| cannot divide by zero
| Numerator = 20
| Denominator = 3
| 20 / 3 = 6.666666666666667
| Numerator = 42x
| ValueError: invalid literal for int() with base 10: '42x'
```

# Catching exceptions – Fractions (III)

`fraction3.py`

```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
    except ValueError:
        print('input not a valid integer')
        continue
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print(f'{numerator} / {denominator} = {result}')
```

catch  
exception

catch  
exception

`Python shell`

```
| Numerator = 5
| Denominator = 2x
| input not a valid integer
| Numerator = 5
| Denominator = 2
| 5 / 2 = 2.5
```





# Catching exceptions – Fractions (IV)

`fraction4.py`

```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
        result = numerator / denominator
        print(f'{numerator} / {denominator} = {result}')
    except ValueError:
        print('input not a valid integer')
    except ZeroDivisionError:
        print('cannot divide by zero')
```



catch  
exceptions

`Python shell`

```
| Numerator = 3
| Denominator = 0
| cannot divide by zero
| Numerator = 3x
| input not a valid integer
| Numerator = 4
| Denominator = 2
| 4 / 2 = 2.0
```

# Keyboard interrupt (*Ctrl-c*)

- throws **KeyboardInterrupt** exception

**infinite-loop1.py**

```
print('starting infinite loop')

x = 0
while True:
    x = x + 1

print(f'done ({x = })')
input('type enter to exit')
```

**Python shell**

```
| starting infinite loop
| Traceback (most recent call last):
|   File 'infinite-loop1.py', line 4, in <module>
|     x = x + 1
| KeyboardInterrupt
```

**infinite-loop2.py**

```
print('starting infinite loop')

try:
    x = 0
    while True:
        x = x + 1
except KeyboardInterrupt:
    pass

print(f'done ({x = })')
input('type enter to exit')
```

**Python shell**

```
| starting infinite loop
| done (x = 23890363) Ctrl-c
| type enter to exit
```

# Keyboard interrupt (*Ctrl-c*)

- Be aware that you likely would like to leave the *Ctrl-c* generated **KeyboardInterrupt** exception unhandled, except when stated explicitly

```
read-int1.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except ValueError: # only ValueError
        continue

print('The value is:', x)

Python shell
| An integer:           Ctrl-c
| KeyboardInterrupt
```

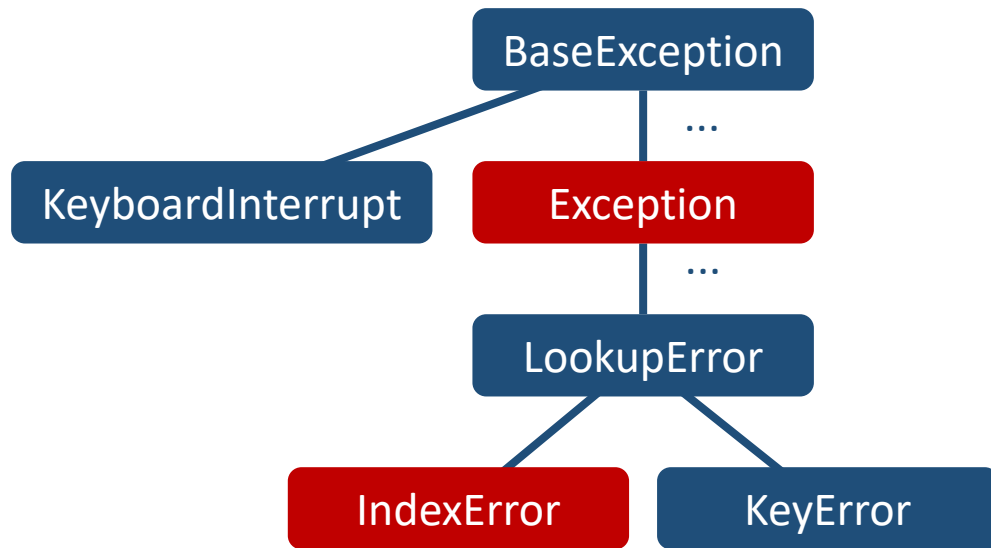
```
read-int2.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except: # all exceptions ← catches KeyboardInterrupt
        continue

print('The value is:', x)

Python shell
| An integer:           Ctrl-c
| An integer:           Ctrl-c
| An integer:
```

- (left) KeyboardInterrupt is unhandled (right) it is handled (intentionally?)

# Exception class hierarchy




`except-twice1.py`

```
try:
    L[4]
except IndexError: # must be before Exception
    print('IndexError')
except Exception:
    print('Fall back exception handler')
```

`except-twice2.py`

```
try:
    L[4]
except Exception: # and subclasses of Exception
    print('Fall back exception handler')
except IndexError:
    print('IndexError') # unreachable
```



# try statement syntax

**try:**

*code*

**except** ExceptionType1:

*code* # executed if raised exception instance of  
# ExceptionType1 (or subclass of ExceptionType1)

**except** ExceptionType2:

*code* # executed if exception type matches and none of  
# the previous except statements matched

...

**else:**

*code* # only executed if no exception was raised

**finally:**

*code* # always executed independent of exceptions  
# typically used to clean up (like closing files)

arbitrary number of except cases

# except variations

`except:` # catch all exceptions



`except ExceptionType:` # only catch exceptions of class ExceptionType  
# or subclasses of ExceptionType

`except (ExceptionType1, ExceptionType2, ..., ExceptionTypek):`  
# catch any of k classes (and subclasses)  
# paranthesis cannot be omitted

`except ExceptionType as e:`  
# catch exception and assign exception object to e  
# e.args contains arguments to the raised exception

# Raising exceptions

- An exception is raised (or thrown) using one of the following (the first being an alias for the second):

```
raise ExceptionType
```

```
raise ExceptionType()
```

```
raise ExceptionType(args)
```

**abstract.py**

```
class A():
    def f(self):
        print('f')
        self.g()

    def g(self):
        raise NotImplementedError

class B(A):
    def g(self):
        print('g')
```

**Python shell**

```
> B().f()
| f
| g
> A().f()
| f
| NotImplementedError
```

# User exceptions

- New exception types are created using **class inheritance** from an existing exception type (possibly defining `__init__`)

tree-search.py

```
class SolutionFound(Exception): # new exception
    pass

def recursive_tree_search(x, tree):
    if isinstance(tree, tuple):
        for child in tree:
            recursive_tree_search(x, child)
    elif x == tree:
        raise SolutionFound # found x in tree

def tree_search(x, tree):
    try:
        recursive_tree_search(x, tree)
    except SolutionFound:
        print('found', x)
    else:
        print('search for', x, 'unsuccessful')
```

Python shell

```
> tree_search(8, ((3,2),5,(7,(4,6))))
| search for 8 unsuccessful
> tree_search(7, ((3,2),5,(7,(4,6))))
| found 7
```



# 3 ways to read lines from a file

## Steps

1. Open file using `open`
2. Read lines from file using
  - a) `for line in filehandler:`
  - b) `filehandler.readlines`
  - c) `filehandler.readline`
3. Close file using `close`

`open ('filename.txt')` assumes the file to be in the same folder as your Python program, but you can also provide a full path  
`open ('c:/Users/gerth/Documents/filename.txt')`

try to open file for reading filename

filehandle

iterate over lines in file

close file when done

```
reading-file1.py
f = open('reading-file1.py')
for line in f:
    print('> ', line, end='')
f.close()
```

read all lines into a list of strings

```
reading-file2.py
f = open('reading-file2.py')
lines = f.readlines()
f.close()
for line in lines:
    print('> ', line, end='')
```

read single line (terminated by '\n')

```
reading-file3.py
f = open('reading-file3.py')
line = f.readline()
while line != '':
    print('> ', line, end='')
    line = f.readline()
f.close()
```

# 3 ways to write lines to a file

- Opening file:

`open(filename, mode)`

where *mode* is a string, either 'w' for opening a new (or truncating an existing file) and 'a' for appending to an existing file

- Write single string:

`filehandle.write(string)`

Returns the number of characters written

- Write list of strings strings:

`filehandle.writeline(list)`

- Newlines ('\\n') must be written explicitly

- `print` can take an optional `file` argument

```
write-file.py
f = open('output-file.txt', 'w')
f.write('Text 1\\n')
f.writelines(['Text 2\\n', 'Text 3 '])
f.close()

g = open('output-file.txt', 'a')
print('Text 4', file=g)
g.writelines(['Text 5 ', 'Text 6'])
g.close()

output-file.txt
Text 1
Text 2
Text 3 Text 4
Text 5 Text 6
```

try to open file for writing

write mode

write single string to file

write list of strings to file

append to existing file

# Exceptions while dealing with files

- When dealing with files one should be prepared to handle errors / raised exceptions, e.g. `FileNotFoundError`

```
reading-file4.py
```

```
try:
    f = open('reading-file4.py')
except FileNotFoundError:
    print('Could not open file')
else:
    try:
        for line in f:
            print('> ', line, end='')
    finally:
        f.close()
```

# Opening files using `with` (recommend way)

- The Python keyword `with` allows to create a *context manager* for handling files
- Filehandle will automatically be closed, also when exceptions occur
- Under the hood: filehandles returned by `open` support `__enter__` and `__exit__` methods

`f` = result of calling `__enter__()`  
on result of `open` expression,  
which is the file handle

reading-file5.py

```
with open('reading-file5.py') as f:  
    for line in f:  
        print('> ', line, end='')
```

# Does a file exist?

- Module `os.path` contains a method `isfile` to check if a file exists

```
checking-files.py
```

```
import os.path

filename = input('Filename: ')
if os.path.isfile(filename):
    print('file exists')
else:
    print('file does not exist')
```

# module `sys`

- Module `sys` contains the three standard file handles  
`sys.stdin` (used by the `input` function)  
`sys.stdout` (used by the `print` function)  
`sys.stderr` (error output from the Python interpreter)

```
sys-test.py
```

```
import sys
sys.stdout.write('Input an integer: ')
x = int(sys.stdin.readline())
sys.stdout.write(f'{x} square is {x ** 2}')
```

```
Python shell
```

```
| Input an integer: 10
| 10 square is 100
```

# print(..., file=output file)

sys-print-file.py

```
import sys

def complicated_function(file):
    print('Hello world', file=file) # print to file or STDOUT

while True:
    file_name = input('Output file (empty for STDOUT): ')

    if file_name == '':
        file = sys.stdout
        break
    else:
        try:
            file = open(file_name, 'w')
            break
        except Exception:
            pass

complicated_function(file)

if file != sys.stdout:
    file.close()
```

# PEP8 on exceptions

- For all try/except clauses, limit the try clause to the absolute **minimum amount of code** necessary
- The class naming convention applies (**CapWords**)
- Use the **suffix "Error"** on your exception names (if the exception actually is an error)
- A bare `except:` clause will catch `SystemExit` and `KeyboardInterrupt` exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use `except Exception:`



# Performance of scanning a file

- Python can efficiently scan through quite big files

File	Size	Time
<u>Atom_chem_shift.csv</u>	≈ 750 MB	≈ 8 sec
<u>cano.txt</u>	≈ 3.7 MB	≈ 0.1 sec

The first search finds all lines related to ThrB12-DKP-insulin (Entry ID 6203) in a chemical database available from [www.bmrb.wisc.edu](http://www.bmrb.wisc.edu)

The second search finds all occurrences of “Germany” in Conan Doyle's complete Sherlock Holmes available at [sherlock-holm.es](http://sherlock-holm.es)

file-scanning.py

```
from time import time
for filename, query in [
    ('Atom_chem_shift.csv', '6203'),
    ('cano.txt', 'Germany')
]:
    count = 0
    matches = []
    start = time()
    with open(filename) as f:
        for i, line in enumerate(f, start=1):
            count += 1
            if query in line:
                matches.append((i, line))
    end = time()

    for i, line in matches:
        print(i, ':', line, end='')
    print('Duration:', end - start)
    print(len(matches), 'of', count, 'lines match')
```

Python shell

```
...
3057752 : 195,,2,2,30,30,THR,HB,H,1,4.22,0.02,,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057753 : 196,,2,2,30,30,THR,HG21,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057754 : 197,,2,2,30,30,THR,HG22,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057755 : 198,,2,2,30,30,THR,HG23,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,228896,6203,2
Duration: 7.760039329528809
329 of 9758361 lines match
57557 :      "Well, then, to the West, or to England, or to Germany, where father
66515 :      kind master. He wanted me to go with his wife to Germany yesterday,
66642 :      of business in Germany in the past and my name is probably familiar
73273 :      associates with Germany. This he placed in his instrument cupboard.
Duration: 0.07700657844543457
4 of 76764 lines match
```

## sudoku.py

```
class Sudoku:
    def __init__(self, puzzle):
        self.puzzle = puzzle

    def solve(self):
        def find_free():
            for i in range(9):
                for j in range(9):
                    if self.puzzle[i][j] == 0:
                        return (i, j)
            return None

        def unused(i, j):
            i_, j_ = i // 3 * 3, j // 3 * 3
            cells = {(i, k) for k in range(9)}
            cells |= {(k, j) for k in range(9)}
            cells |= {(i, j) for i in range(i_, i_ + 3)
                       for j in range(j_, j_ + 3)}
            return set(range(1, 10)) - {self.puzzle[i][j] for i, j in cells}

        class SolutionFound(Exception):
            pass

        def recursive_solve():
            cell = find_free()
            if not cell:
                raise SolutionFound
            i, j = cell
            for value in unused(i, j):
                self.puzzle[i][j] = value
                recursive_solve()
            self.puzzle[i][j] = 0

        try:
            recursive_solve()
        except SolutionFound:
            pass
```

## sudoku.py (continued)

```
    def print(self):
        for i, row in enumerate(self.puzzle):
            cells = [f' {c} ' if c else ' . ' for c in row]
            print(''.join([''.join(cells[j:j+3]) for j in (0,3,6)]))
            if i in (2, 5):
                print('-----+-----+-----')

        with open('sudoku.txt') as f:
            A = Sudoku([[int(x) for x in line.strip()] for line in f])

        A.solve()
        A.print()
```

## sudoku.txt

```
517600034
289004000
346205090
602000010
038006047
000000000
090000078
703400560
000000000
```

## Python shell

```
| 5 1 7 | 6 9 8 | 2 3 4
| 2 8 9 | 1 3 4 | 7 5 6
| 3 4 6 | 2 7 5 | 8 9 1
|-----+-----+-----|
| 6 7 2 | 8 4 9 | 3 1 5
| 1 3 8 | 5 2 6 | 9 4 7
| 9 5 4 | 7 1 3 | 6 8 2
|-----+-----+-----|
| 4 9 5 | 3 6 2 | 1 7 8
| 7 2 3 | 4 8 1 | 5 6 9
| 8 6 1 | 9 5 7 | 4 2 3
```

# match – case (since Python 3.10)

- Assume we want to do different things depending on the value of an expression (different *cases*)
- Can be done using `if`, but also using `match – case`, that is also evaluated top-down

match-case.py

```
x = 7
if x == 1:
    print('x is one')
elif x == 2:
    print('x is two')
elif x == 3 or x == 4 or x == 5:
    print('x is three, four or five')
else:
    print(x, 'is not in the range 1-5')
```

Python shell

```
| 7 is not in the range 1-5
```

match-case.py

```
x = 7
match x: # match expression
    case 1:
        print('x is one')
    case 2:
        print('x is two')
    case 3 | 4 | 5: # match any of the cases
        print('x is three, four or five')
    case value: # else, value = variable name
        print(value, 'is not in the range 1-5')
```

Python shell

```
| 7 is not in the range 1-5
```

# match - case

Can match...

- simple values
- named variable values
- guards (`if`)
- sequences of values
- dictionaries
- builtin types
- user defined classes
- nested structures of the above

match-case.py

```
class Color:
    RED = 'ff0000'
    GREEN = '00ff00'
    BLUE = '0000ff'

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'Point({self.x}, {self.y})'
```

match-case.py

```
def f(x):
    match x:
        case 42:
            return 'the integer 42'
        case 1 | 2 | 3 | 4 | 5:
            return 'integer in range(1, 6)'
        case (1, 2):
            return 'sequence containing the elements 1 and 2'
        case [x, 2]:
            return 'sequence of length 2, last=2, first=' + str(x)
        case (x, y) if x + y == 7: # guard
            return 'sequence with two values with sum 7'
        case [0, 1, *x]: # x is list of remaining elements in sequence
            return 'sequence starting with 0 and 1, and tail ' + str(x)
        case {'a': 7, 'b': x}:
            return 'dictionary "a" -> 7, "b" -> ' + str(x)
        case (('a' | 'b'), ('c' | 'd')):
            return 'tuple length 2, first "a" or "b", last "c" or "d"'
        case (('x' | 'y') as fst, ('x' | 'y') as snd):
            return '(fst, snd), where fst=' + str(fst) + ', snd=' + str(snd)
        case float(value): # test on builtin type
            return 'a float ' + str(value)
        case Color.RED: # class or object attribute
            return 'the color red'
        case Point(x=7, y=value): # Point object with attributes x and y
            return 'a Point object with x=7, and y=' + str(value)
        case Point(x, y): # requires __match_args__ in class Point
            return 'a point Point(' + str(x) + ', ' + str(y) + ')'
        case e: # ⚠ using the wildcard _ would not bind to a variable
            return 'cannot match ' + repr(e)
```

## Python shell (match-case.py continued)

```
> for x in [42, 1, [1, 2], [7, 2], range(3, 5), (3, (5, 7)), (0, 1, 2, 3, 4, 5), {'a':7, 'b':42, 'c':1},
           ('b', 'c'), ('y', 'x'), 3.14, 'ff0000', Point(7, 42), Point(3, 5), 'abc']:
    print('f(' + repr(x) + ') = ' + repr(f(x)))
f(42) = 'the integer 42'
f(1) = 'integer in range(1, 6)'
f([1, 2]) = 'sequence containing the elements 1 and 2'
f([7, 2]) = 'sequence of length 2, last=2, first=7'
f(range(3, 5)) = 'sequence with two values with sum 7'
f((3, (5, 7))) = 'a triplet (3, (5, 7))'
f((0, 1, 2, 3, 4, 5)) = 'sequence starting with 0 and 1, and tail [2, 3, 4, 5]'
f({'a': 7, 'b': 42, 'c': 1}) = 'dictionary "a" -> 7, "b" -> 42'
f(('b', 'c')) = 'tuple length 2, first "a" or "b", last "c" or "d"'
f(('y', 'x')) = '(fst, snd), where fst=y, snd=x'
f(3.14) = 'a float 3.14'
f('ff0000') = 'the color red'
f(Point(7, 42)) = 'a Point object with x=7, and y=42'
f(Point(3, 5)) = 'a point Point(3, 5)'
f('abc') = "cannot match 'abc'"
```