**Using exceptions**

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

https://docs.python.org/3/reference/executionmodel.html#exceptions

https://nedbatchelder.com/text/exceptions-vs-status.html   
https://nedbatchelder.com/text/exceptions-in-the-rainforest.html

**INTRODUCTION #do osbługi błędów**

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.

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'.

Broadly speaking, there are two ways to handle errors as they pass from layer to layer in software: throwing exceptions and returning status codes.

Why are we using exceptions [instead of error codes]?   
(a) To raport errors or warnings [both methods].   
(b) Exceptions are propagated by default up the call stack (error codes do not). The intermediate layers of code may have no knowledge of them (clean code). – domyslnie się propagują  
(c) An exception class hierarchy can be used for related results. – łapiemy całą klasę wyjątków  
(d) Exceptions cannot be ignored (unless we want this), return codes can.   
(e) Exceptions can carry richer information than error codes.   
(f) Exceptions leave the primary channel (function returns) available for the primary work. Status returns can’t even be used with some functions [C++]. – kody powrotu zwracaja liczby jako return

+----------------------------------------+---------------------+

| Statements | Meaning |

+----------------------------------------+---------------------+

| raise MyException("message") | raising exceptions |rzucanie 😊

| try/except/finally | handling exceptions |

Jakie wyjątki chcemy zlapac i jak je obsłużyć

| class MyException(Exception): pass | creating exceptions |

| assert expression | assertions |

| with ContextManager() as context: pass | context managers |

+----------------------------------------+---------------------+

The built-in exceptions can be generated by the interpreter or built-in functions.

Wbudowane wyjątki:

print(23 / 0) # ZeroDivisionError – Error jeśli wyjątek jest błędem

L = [] # an empty list

print(L[5]) # IndexError

D = {} # an empty dictionary

print(D["key"]) # KeyError

**EXCEPTION HIERARCHY**

>>> import exceptions # Py2.6+

>>> help(exceptions)

>>> import builtins # Py3, built-in functions, exceptions, and other objects

>>> help(builtins)

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception # all other current built-in exceptions

+-- StopIteration

+-- StandardError

| +-- ArithmeticError

| +-- AssertionError

| +-- AttributeError

| +-- NameError

| +-- SyntaxError

| +-- IndentationError

| +-- TypeError

| +-- ValueError

| +-- ImportError

| +-- MemoryError

| +-- RuntimeError

| +-- NotImplementedError

| +-- EnvironmentError

| +-- IOError

| +-- OSError

| +-- LookupError

| +-- IndexError

| +-- KeyError

| +-- ...

+-- Warning

+-- DeprecationWarning

+-- UserWarning

+-- SyntaxWarning

+-- RuntimeWarning

+-- ...

**raise statements – rzucanie wyjątków**

https://docs.python.org/3/reference/simple\_stmts.html#raise

**INTRODUCTION**

The 'raise' statement is used to explicitly raise an exception. A traceback object is normally created automatically when an exception is raised and attached to it as the '\_\_traceback\_\_' attribute, which is writable.

# Syntax.

#

# raise exception\_instance

# raise exception\_class

#

# Using the context of the old exception (Py3)

#

# raise new\_exception from old\_exception

# raise new\_exception from None # suppress exception chaining

raise IndexError()

raise IndexError("message") # recommended

raise IndexError

raise # re-raise the last exception that was active in the current scope

**EXAMPLES**

def power3(x):

if x < 0:

raise ValueError("negative x")

return x \* x \* x

**try statements – łapanie wyjątków**

https://docs.python.org/3/reference/compound\_stmts.html#try

**INTRODUCTION**

The 'try' statement can be used used to specify exception handlers and/or cleanup code for a group of statements.

The 'except' clause(s) specify one or more exception handlers.

The 'finally' clause 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.

# Syntax.

# (1) try + except (one or more) + else (optional) + finally (optional)

# (2) try + finally

try: - wrap tam gdzie spodziewamy się wyjątków

statements

except ExceptionClass1: # matching ExceptionClass1 – except łapie wyjątki

statements

except (ExceptionClass2, ExceptionClass3): # matching two exception classes

statements

except ExceptionClass4 as exception1:

# exception1 is an instance of ExceptionClass4

statements

except (ExceptionClass4, ExceptionClass5) as exception2:

statements

#except: # if present, must be the last; it matches any exception

#puste except łapie wszystkie wyjątki (nawet SysExit czy KeyboardInterrupt

except Exception: # recommended – lepiej takie

statements

else: # optional, executed if no exception was raised

statements

finally: # if present, it is always executed-zawsze wykonane!

statements

L = []

# L = [1, 2, 3]

try:

print(L[1])

except IndexError:

print("exception is present")

else:

print("no exception")

print("after try")

afile = open("book.txt")

try: # working with the file

text = afile.read() # reading the file

finally: # the file will always be closed

afile.close()

print("after try")

# Now 'with' is recommended to open and close the file – przypomnij!

import sys

print(sys.exc\_info()) # return (type, value, traceback)

# Return information about the most recent exception caught by an except

# clause in the current stack frame or in an older stack frame.

**EXAMPLES**

# Py2 and Py3 compatibility.

try:

input = raw\_input # input() in Py2 is slow and dangerous

except NameError: # we are in Py3

pass

nieważne, czy py2 pczy py3 to i tak zadziała

# Py2 and Py3 compatibility.

try:

integer\_types = (int, long)

except NameError: # we are in Py3

integer\_types = (int,)

# Usage: isinstance(variable, integer\_types)

# Interactions with users.

while True:

reply = input("Enter int: ") # Py3

try:

value = int(reply)

except ValueError:

print("This is not int!")

else:

break

print("value is set to {}".format(value))

**User-defined exceptions**

https://docs.python.org/3/tutorial/errors.html#tut-userexceptions

**INTRODUCTION**

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the 'Exception' class, either directly or indirectly.

All exceptions must be instances of a class that derives from 'BaseException'.

# Syntax.

class SomeError(Exception): # a name with 'Error' if we want to raport errors

docstring # optional

statements

class MyError(Exception):

pass # the default \_\_init\_\_() is used

raise MyError("message") # raising the exception

exception = MyError("a", "b") # creating an instance of the exception class

print(exception.args) # ('a', 'b'), the tuple is created by the constructor

print(exception) # ('a', 'b'), str() uses exception.args

**assert statements**

https://docs.python.org/3/reference/simple\_stmts.html#the-assert-statement

**ASSERTIONS**

Assert statements are a convenient way to insert debugging assertions into a program.

# Syntax.

# The name \_\_debug\_\_ is true if Python was not started with an -O option.

# Try 'python3 --help' in the shell to see other options.

assert expression - prawdziwne

# It is equivalent to

# if \_\_debug\_\_:

# if not expression: raise AssertionError

assert expression1, expression2 – sprawdzamy pierwsze, jeśli faływe to AssestionError

# It is equivalent to

# if \_\_debug\_\_:

# if not expression1: raise AssertionError(expression2)

assert False # AssertionError

assert False, 'message' # AssertionError: message

assert False, 2 + 2 # AssertionError: 4

**Iterators**

https://docs.python.org/3/tutorial/classes.html

https://docs.python.org/3/library/functions.html#iter

https://docs.python.org/3/library/stdtypes.html#typeiter

**INTRODUCTION**

Three notions:   
(1) an iterable is anything that you can iterate over [iter() can build an iterator],   
(2) an iterator is the thing that does the actual iterating [\_\_iter\_\_() and \_\_next\_\_() methods should be present],   
(3) generators are one of the simpler ways to create your own iterators ['lazy evaluation'; generator expressions and generator functions].

for item in iterable: # iteration

print(item)

L = list(iterable)

result = sum(iterable)

high = max(iterable)

low = min(iterable)

**FINITE ITERATORS**

# Builtin function.

reversed(sequence) # reversed object

# Generator expression.

sequence = "word"

gen = (sequence[i] for i in range(len(sequence)-1,-1,-1)) # generator object

# Generator function.

def Reverse(sequence): # a generator iterator is returned

"""A reverse iterator based on a sequence."""

idx = len(sequence)

while idx > 0:

idx = idx-1

yield sequence[idx]

# The other version based on a class.

class Reverse: # an instance of this class is an iterator

"""A reverse iterator based on a sequence."""

def \_\_init\_\_(self, sequence):

self.sequence = sequence

self.idx = len(sequence)

def \_\_iter\_\_(self): # for the iter() function

return self

def \_\_next\_\_(self): # for the next() function

if self.idx == 0:

raise StopIteration

self.idx -= 1

return self.sequence[self.idx]

next = \_\_next\_\_ # compatibility

# Python 2 has X.next().

# Python 3 has X.\_\_next\_\_().

# Python 3 i 2.6+ have a builtin function next(X).

# Usage (for both versions).

for char in Reverse("spam"):

print(char) # m, a, p, s

for item in Reverse([1, 2, 3, 4]):

print(item) # 4, 3, 2, 1

**ITER**

# iter(iterable) return iterator

# iter(callable, sentinel) return iterator

# an iterator based on a sequence

it1 = iter([1, 2]) # list\_iterator object in Py3 (lists have \_\_iter\_\_)

it2 = iter("abcd") # str\_iterator object in Py3 (strings have \_\_iter\_\_)

it3 = iter(it2) # all iterators are also iterables

assert it3 is it2

# Once you have an iterator, the only thing you can do with it is get

# its next item. And you’ll get a stop iteration exception if you ask

# for the next item but there aren’t anymore items.

print(next(it1)) # 1

print(next(it1)) # 2

print(next(it1)) # StopIteration

class Dumb:

def \_\_init\_\_(self, sequence):

self.data = sequence

def \_\_getitem\_\_(self, n):

return self.data[n]

x = Dumb("word")

print(x[1]) # o

#print(x[10]) # IndexError: string index out of range

#len(x) # AttributeError: Dumb instance has no attribute '\_\_len\_\_'

for item in x: # iteration

print(item)

# manual iteration

y = iter(x) # make an iterator, iter() use the sequence protocol (\_\_getitem\_\_)

print(next(y)) # w

print(next(y)) # o

print(next(y)) # r

print(next(y)) # d

print(next(y)) # StopIteration, the iterator is exhausted

# Chunking list by two using an iterator.

it = iter(range(6))

list(zip(it, it)) # [(0, 1), (2, 3), (4, 5)]

**FOR LOOP EXPLAINED**

# https://www.programiz.com/python-programming/iterator

for element in iterable:

# do something with element

process(element)

# It is equivalent to the following code.

# create an iterator object from that iterable

iter\_obj = iter(iterable)

# infinite loop

while True:

try:

# get the next item

element = next(iter\_obj)

# do something with element

process(element)

except StopIteration:

# if StopIteration is raised, break from loop

break

**AN ITERATOR BASED ON A CALLABLE OBJECT**

**Collable- nie jest to funkcja, można wywołać z nawiasami bez argumentów**

class MyInt:

def \_\_init\_\_(self):

self.n = -1

def \_\_call\_\_(self):

self.n = self.n + 1

return self.n

x = MyInt() # x is a MyInt instance, a callable object

print(x()) # returns 0

print(x()) # returns 1

print(x()) # returns 2

# Preparing an iterator 'y'.

# If the callable returns 5 then y stops.

y = iter(MyInt(), 5) # y is a callable-iterator object

for i in y: # returns 0, 1, 2, 3, 4 (without 5)

print(i)

# 5 was used to stop the iterator y.

print(next(y)) # StopIteration is raised, y is exhausted

# Infinite iterators.

import itertools

iter\_zeros1 = iter(int, 1) # int() gives 0

iter\_zeros2 = iter((lambda: 0), 1)

iter\_zeros3 = itertools.cycle([0]) # itertools.cycle(sequence)

iter\_zeros4 = itertools.repeat(0) # itertools.repeat(x, times=5)

**Using itertools**

https://docs.python.org/3/library/itertools.html

https://docs.python.org/3/library/itertools.html#itertools-recipes

https://realpython.com/python-itertools/

**INTRODUCTION**

This module implements functions creating iterators for efficient looping.

The real power of 'itertools' lies in composing the functions to create fast, memory-efficient, and good-looking code.

import itertools

# Infinite iterators

# itertools.count(start=0, step=1])

itertools.count(10) # yield 10 11 12 13 14 ...

itertools.count(10, 5) # yield 10 15 20 25 ...

# enumerate(iterable, start=0) emulator without for loop

zip(itertools.count(start=0), iterable)

# itertools.cycle(iterable)

itertools.cycle('ABCD') # yield A B C D A B C D ...

itertools.cycle(range(1, 4)) # yield 1 2 3 1 2 3 1 2 3

# itertools.repeat(item [,n])

itertools.repeat(10) # yield 10 10 10 10 ...

itertools.repeat(10, 3) # yield 10 10 10 (3 times)

list(map(pow, range(10), itertools.repeat(2))) # works for any range (no for loop)

#list(map(pow, range(10), iter((lambda: 2), 1))) # the same

#list(pow(x, 2) for x in range(10)) # simple

# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# itertools.chain(\*iterables)

itertools.chain('ABC', 'DEF') # yield A B C D E F

# itertools.chain.from\_iterable(iterable) # alternate constructor for chain()

itertools.chain.from\_iterable(['ABC', 'DEF']) # yield A B C D E F

# itertools.islice(iterable, stop)

# itertools.islice(iterable, start, stop [, step])

# If 'stop' is None, then iteration continues until the iterator is exhausted, if at all.

list(itertools.islice('ABCDEFG', 2)) # ['A', 'B'], stop=2

list(itertools.islice('ABCDEFG', 2, None)) # ['C', 'D', 'E', 'F', 'G']

list(itertools.islice('ABCDEFG', 0, None, 2)) # ['A', 'C', 'E', 'G']

# itertools.zip\_longest(\*iterables[, fillvalue]) in Py3

# itertools.izip\_longest(\*iterables[, fillvalue]) in Py2.6+

# It can be used for polynomials.

itertools.zip\_longest('ABCD', 'xy', fillvalue='-')

# yield ('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')

itertools.zip\_longest([10, 20], [1, 2, 3, 4], fillvalue=0)

# yield (10, 1), (20, 2), (0, 3), (0, 4)

# itertools.product(\*iterables, repeat=1)

# Cartesian product of input iterables.

# product(A, B) returns the same as ((x,y) for x in A for y in B).

list(itertools.product("abc", "123"))

# [('a', '1'), ('a', '2'), ('a', '3'),

# ('b', '1'), ('b', '2'), ('b', '3'),

# ('c', '1'), ('c', '2'), ('c', '3')]

[x+y for (x,y) in itertools.product("abc","123")]

# ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']

[x+y for (x,y) in itertools.product("abc", repeat=2)]

# ['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']

# Combinatoric generators.

# itertools.permutations(p[, r])

# yield r-length tuples, all possible orderings, no repeated elements

list(itertools.permutations([1, 2, 3])) # 3! = 6 perms

# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations([1, 2, 3], 2))

# [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]

# itertools.combinations(p, r)

# yield r-length tuples, in sorted order, no repeated elements

list(itertools.combinations([1, 2, 3, 4], 3))

[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]

# itertools.combinations\_with\_replacement(p, r)

# yield r-length tuples, in sorted order, with repeated elements

list(itertools.combinations\_with\_replacement([1, 2, 3, 4], 3))

# [(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4), (1, 2, 2), (1, 2, 3),

# (1, 2, 4), (1, 3, 3), (1, 3, 4), (1, 4, 4), (2, 2, 2), (2, 2, 3),

# (2, 2, 4), (2, 3, 3), (2, 3, 4), (2, 4, 4), (3, 3, 3), (3, 3, 4),

# (3, 4, 4), (4, 4, 4)]

**EXAMPLES**

# Given a list of values inputs and a positive integer n,

# write a function that splits inputs into groups of length n.

def naive\_grouper(inputs, n):

num\_groups = len(inputs) // n

return [tuple(inputs[i\*n:(i+1)\*n]) for i in range(num\_groups)]

# all items from inputs are in the memory!

def better\_grouper(inputs, n):

iters = [iter(inputs)] \* n

# it is a list of n references to the same iterator

return zip(\*iters)

# zip returns an iterator over tuples

def best\_grouper(inputs, n, fillvalue=None):

iters = [iter(inputs)] \* n

return itertools.zip\_longest(\*iters, fillvalue=fillvalue)

# it works even if n is not a factor of the length of inputs

**Linked lists with iterators**

https://en.wikipedia.org/wiki/Linked\_list

**INTRODUCTION**

A linked list is a linear collection of items.

class Node:

def \_\_init\_\_(self, data=None, next=None):

self.data = data

self.next = next

def \_\_str\_\_(self):

return str(self.data)

class SingleList:

def \_\_init\_\_(self):

self.head = None

self.tail = None

def is\_empty(self):

return self.head is None

def insert\_head(self, data): # O(1) time

node = Node(data)

if self.is\_empty():

self.head = self.tail = node

else:

node.next = self.head

self.head = node

def insert\_tail(self, data): # O(1) time

node = Node(data)

if self.is\_empty():

self.head = self.tail = node

else:

self.tail.next = node

self.tail = node

def remove\_head(self): # O(1) time

if self.is\_empty():

raise ValueError("empty list")

node = self.head

if self.head == self.tail:

self.head = self.tail = None

else:

self.head = self.head.next

node.next = None # cleaning

return node.data

# Usage.

SL = SingleList() # head = tail = None

SL.insert\_head(11) # [11]

# head=tail

# +----++

# | 11 ||

# +----++

SL.insert\_head(22) # [22, 11]

# head tail

# +----++ +----++

# | 22 |+--o| 11 ||

# +----++ +----++

SL.insert\_tail(33) # [22, 11, 33]

# head tail

# +----++ +----++ +----++

# | 22 |+--o| 11 |+--o| 33 ||

# +----++ +----++ +----++

while not SL.is\_empty(): # order 22, 11, 33

print(SL.remove\_head())

**ITERATORS**

# Solution 1.

# SL = SingleList()

# assert not isinstance(iter(SL), SingleList)

# assert id(iter(SL)) != id(SL)

class SingleList:

# ... other methods ...

def \_\_iter\_\_(self): # using a generator function

node = self.head

while node:

yield node.data

node = node.next

# Solution 2.

# SL = SingleList()

# assert isinstance(iter(SL), SingleList)

# assert id(iter(SL)) == id(SL)

class SingleList:

# ... other methods ...

def \_\_iter\_\_(self):

self.current = self.head

return self

def \_\_next\_\_(self):

if self.current:

node = self.current

self.current = self.current.next

return node.data

else: # self.current is None

raise StopIteration

next = \_\_next\_\_ # compatibility

# Usage.

SL = SingleList()

SL.insert\_head(11) # [11]

SL.insert\_head(22) # [22, 11]

SL.insert\_tail(33) # [22, 11, 33]

for item in SL: # order 22, 11, 33

print(item)

**with statements**

PEP 343 - The “with” statement

https://docs.python.org/3/library/stdtypes.html#context-manager-types

https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers

**INTRODUCTION**

Python’s 'with' statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed (\_\_enter\_\_) and exited when the statement ends (\_\_exit\_\_).

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

The 'with' statement allows common 'try + except + finally' usage patterns to be encapsulated for convenient reuse.

# Syntax.

with expression as target: # 'as target' is optional

statements

with A() as a, B() as b:

statements

# is semantically equivalent to nested 'with' statements:

with A() as a:

with B() as b:

statements

with open('book.txt', 'r') as infile: # infile will be closed

text = infile.read()

with open('results.txt', 'w') as outfile: # outfile will be closed

outfile.write("message\n")

**CONTEXT MANAGERS**

class ContextManager:

def \_\_init\_\_(self): # optional

pass

def message(self, argument):

print("processing {}".format(argument))

def \_\_enter\_\_(self):

print("start with")

return self # or something else;

# the return value will be assigned to 'target'

def \_\_exit\_\_(self, exception\_type, exception\_value, exception\_traceback):

if exception\_type is None: # (None, None, None) is supplied to \_\_exit\_\_

print("normal exit") # no exception

# put cleanup code here

return True

else: # the exception type, value, traceback are supplied to \_\_exit\_\_

print("exception raised is {}".format(exception\_type))

# put cleanup code here

return False # exception is reraised

# return True # exception is suppressed

# Usage.

with ContextManager() as context:

context.message("test 1")

print("point reached")

with ContextManager() as context:

context.message("test 2")

raise TypeError

print("point not reached")

print("after with statements")

# Results.

# start with

# processing test 1

# point reached

# normal exit

# start with

# processing test 2

# exception raised is <class 'TypeError'>

Homework7

7.1 – zwrócić przez iter?

7.2 iteratorek xD

Użyć: albo itertools albo sami zbudować podanymi narzędziami