HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW13

Generators
Decorators
Exceptions

Information Technology

May 15, 2025

Content

- Try-Except
  - Concept
  - Simple Example
  - Error Tree (link to documentation)
  - Own Errors
- Assert Statement
- Generator
  - Iterator
  - Concept of Generator
  - Syntax
  - Typical Use Cases
- Decorator
  - From Function Reference to Decorator
  - Examples

Try-Except

Information Technology

May 15, 2025

Try-Except: Concept

Under certain conditions, the interpreter is unable to execute particular expressions.

Interpret the following quote:

*“Measuring programming progress
by lines of code is like
measur ng a rcraft bu ld ng progress
by we ght.”*

Bill Gates, Co-Founder Microsoft

Try-Except: Concept

When the interpreter struggles executing certain instructions, it returns a feedback in the form of an ERROR and breaks the program execution immediately.

Such errors may result from:

- Programming mistakes
- Connection issues
- Inappropriate type matching
- Out of bound list access
- Missing files or references

```
li = [1,2,3,4,5]      # li with 5 items
num = li[len(li)]     # li[5]
        IndexError: list index out of range

print(f'Hello number {num}.')
```

In order to make the application more robust, Python supports the concept of try-except (catch).

- Try-except allows to react on **raised** errors.

Try-Except: Concept

The try-except clause allows to catch a particular or a bunch of exceptions. In this way, the program gets the ability to handle exceptions and treat the unexpected situation accordingly.

The try-except clause comprises 4 blocks with **strict** order:

- `try`: initiates the risky code section
- `except`: handling of exception (optional if finally block available)
- `else`: interpreted when no error was raised
- `finally`: always interpreted at the end of the clause (optional if except block available)

```
try:
    some expression
    some expression
except:
    some handling
else:
    if no exception
finally:
    final statements
```

Without `except`, the application remains unstable.

- only the **`except`** block is able to react on raised errors.

Try-Except: Simple Example

The following method separately handles two specific (`ValueError` and `ZeroDivisionError`) and all other errors raised for irregular input.

```
try:
    num = input('Enter a number: ')
    frac = 1/int(num)
except ValueError as e:
    print('no number given:', e)
except ZeroDivisionError:
    print('can not divide by 0')
except:
    print('any other error happend')
else:
    print(f'the fraction is: {frac}')
finally:
    print('leaving try-except clause')
```

Try-Except: Error Tree

Catching and handling multiple error types in the same way can be achieved in two different ways:

By **explicit** combination in a tuple:

```
try:
    num = input('Enter a number: ')
    frac = 1/int(num)
except (ValueError, ZeroDivisionError) as e:
    print('error is:', e)
else:
    print(f'the fraction is: {frac}')
finally:
    print('leaving try-except clause')
```

By **implicit** combination by error group:

```
try:
    num = input('Enter a number: ')
    frac = 1/int(num)
except Exception as e:
    print('error is:', e)
else:
    print(f'the fraction is: {frac}')
finally:
    print('leaving try-except clause')
```

The hierarchy of the exception depends on the class inheritance. For built-in exceptions, the hierarchy is given in the doc:

- Doc: https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Try-Except: Customize and Raise Errors

Errors can also be customized and raised by the programmer.

Customizing errors implies declaring a new **class** that:

- **inherits** from `Exceptions` or any built-in Error/Warning type.
- Does **not** inherit from class `BaseException`.
- (optionally) customized error message by instance variable `error`.

```
class CustError(Exception):
    def __init__(self, message):
        super().__init__(message)

        # customize error code...
        self.errors = "custom error"
```

An error can be raised at any position in the script using the keyword: **raise**

- optionally, when raising an error, a customized message can be passed.

```
if input('enter number: ') == '0':
    raise CustError('customized error message')
```

Assert Statement

Information Technology

May 15, 2025

Assert Statement

> "it's easier to ask forgiveness than permission"
>
> Grace Murray Hoppers

docs.python.org: “Assert statements are a convenient way to insert **debugging** assertions into a program”

```
x = 21
assert isinstance(x, int), 'Value should be of type int'
```

- `asserts` can be useful for sanity checks in development, testing, and debugging phase.
- `asserts` can get optimized away in production code.
- Use proper exception handling instead.

```
try:
    div = x/27*5+1
except (AttributeError, TypeError):
      raise AssertionError('Value should be of type int')
```

Docu: https://docs.python.org/3/reference/simple_stmts.html#assert

Generator

Information Technology

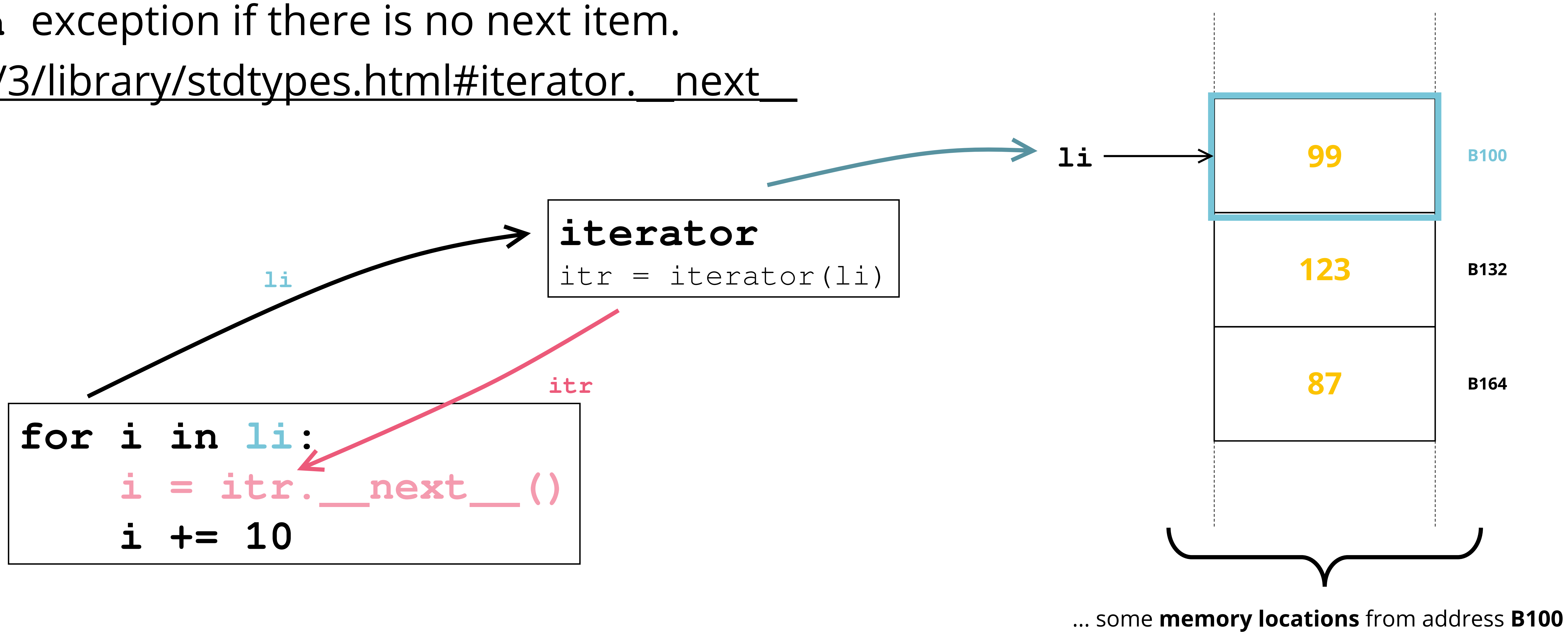
May 15, 2025

Generator: Iterator

Arranging objects in a sequence like a list or tuple enables accessing each element sequentially by iterating over the sequence. Such a functionality, however, requires an **iterator object** that refers to each element sequentially.

An iterator is an object that implements the built-in function __**next**__**()**. The iterator ...

- ... knows the current position and a reference to the next item when called.
- ... raises the `StopIteration` exception if there is no next item.
- Doc: https://docs.python.org/3/library/stdtypes.html#iterator.__next__



Generator: Concept of Generator

For a similar behavior, programmer can create a **generator function**.

Generator functions return a **generator** (iterator) that:

- returns the **next value** of a series when its next function is called.
- memorizes the last position and internal values when returned a value.
- returns values with the keyword `yield` instead of `return`.

call generator function

```
generator_function()
    expression
    expression
    yield
```

returns **generator** (iterator)

```
g_fun = generator_function()
v1 = next(g_fun)          #v1 = 10
v1 = next(g_fun)          #v1 = 20
```

Generator: Syntax

The generator function is like a regular function, yet using the `yield` keyword instead of **`return`**.

A generator function:

- can have parameters.
- returns a **generator** when called.
- can have multiple yields.
- breaks at each yield.
- raise a **`StopIteration`** error when exhausted.
- can be stopped using iterators `close()` method.

```python
def my_gen():
    i = 0
    while i < 5:
        yield i
        i+=1

g = my_gen()
```

Similar to list comprehensions, generator functions can be declared as **generator expressions**:

- with same properties as generator functions, but
- without the keyword yield and are
- declared like list comprehensions yet with round brackets `(expression)`.

```python
g = (i for i in range(5))
```

Generator: Typical Use Cases

Generator functions are typically used for large **deterministic** sequences without keeping the whole sequence in memory. The elements are calculated (produced) one by one on **demand**. This is called “lazy evaluation” and makes generators incredibly **efficient**.

This is very helpful for:

- Non-incremental Iteration over large lists.
- Ring-buffer element indexing.
- Labeling records without complex internal value memory.
- Looking for a particular element in a large sequence (early stopping).

```
>>> k = range(1000000)
>>> any(x<0 for x in k)
False
```

Decorator

Information Technology

May 15, 2025

Decorator: From Function Reference to Decorator

A function can be called from another function:

```
def fnc_1(x):
    return x**2

def fnc_2(xi):
    xi += 10
    return fnc_1(xi)

print(fnc_1(10))        # output: 100
print(fnc_2(10))        # output: 400
```

Assuming, the function called by fnc_2 should be customizable. This can be achieved by a 2nd parameter:

```
def fnc_1(x):
    return x**2

def fnc_2(fx, xi):
    xi += 10
    return fx(xi)

print(fnc_1(10))          # output: 100
print(fnc_2(fnc_1, 10))   # output: 400
```

Decorator: From Function Reference to Decorator

For more convenience, function **`fnc_2`** should have the same interface as **`fnc_1`**.

- A third function **`merge`** installs by reference an arbitrary function (in this case **`fnc_1`**) in **`fnc_2`**.

```
def fnc_1(x):
    return x**2

def fnc_2(fx, xi):
    xi += 10
    return fx(xi)
```

```
def fnc_1(x):
    return x**2

def merge(fx):
    def fnc_2(xi):
        xi += 10
        return fx(xi)
    return fnc_2

f2 = merge(fnc_1)
print(fnc_1(10))       # output: 100
print(f2(10))          # output: 400
```

Decorator: From Function Reference to Decorator

Now we can apply the function **`merge`** as decorator:

```
def fnc_1(x):
    return x**2

def merge(fx):
    def fnc_2(xi):
        xi += 10
        return fx(xi)
    return fnc_2

f2 = merge(fnc_1)
print(fnc_1(10))        # output: 100
print(f2(10))           # output: 400
```

```
def merge(fx):
    def fnc_2(xi):
        xi += 10
        return fx(xi)
    return fnc_2

@merge
def fnc_1(x):
    return x**2

print(fnc_1(10))        # output: 400
```

Decorator: Examples

Following two examples for frequently used decorators:

Execution time measure of particular functions:

```
def deco_timer(fnc):
    from time import time

    def wrapper(*args, **kwargs):
        t1 = time()
        fnc_res = fnc(*args, **kwargs)
        return fnc_res, time()-t1
    return wrapper

@deco_timer
def fnc_1(x):
    return x**1000000

res, t_elapsed = fnc_1(1000000)
print(f'{t_elapsed=}')
print(f'{res}')
```

Properties: a property object has getter, setter, and deleter methods usable as decorators.

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

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch