



Basic Programming

Lesson 04

Outline

1. Function
2. Docstring
3. Comment
4. Function Argument
5. Handle Exception

Function

Defining Functions

```
>>> def square(x):  
...     return x * x  
...  
>>> square(5)  
25  
>>> def launch_missiles():  
...     print("Missiles launched!")  
...  
>>> launch_missiles()  
Missiles launched!  
>>>
```

Early Return

```
>>> def even_or_odd(n):  
...     if n % 2 == 0:  
...         print("even")  
...         return  
...     print("odd")  
...  
>>> w = even_or_odd(31)  
odd  
>>> w is None  
True  
>>>
```

`__feature__`

Hard to pronounce!

dunder

Our way of pronouncing special names

A portmanteau of "double underscore"

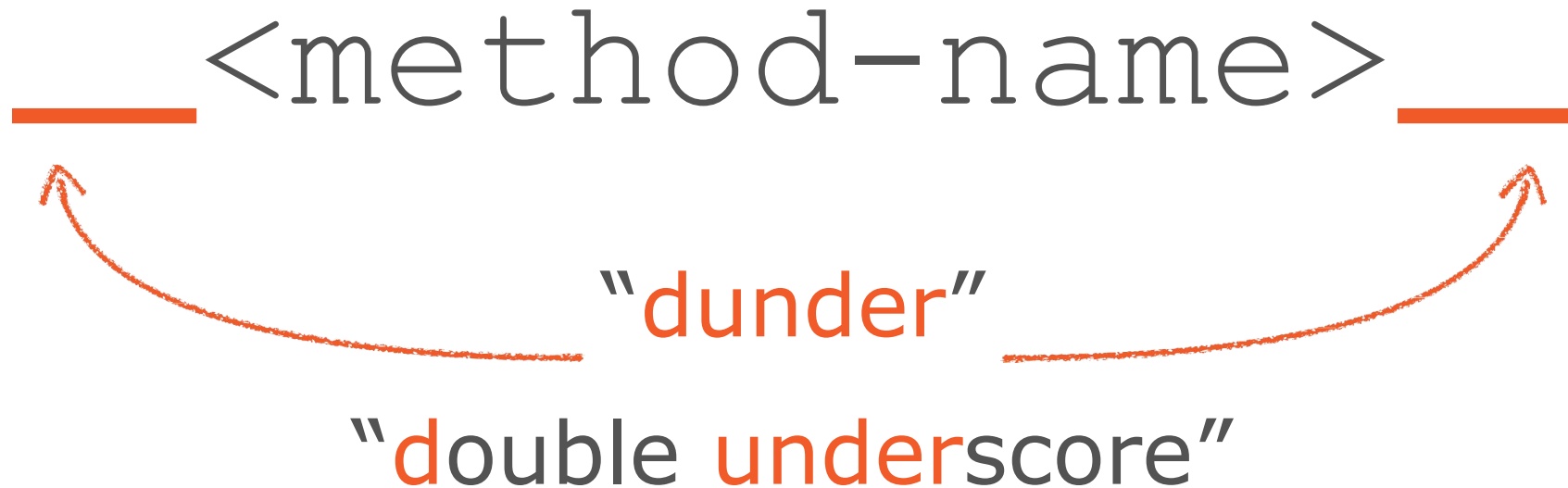
Instead of "underscore underscore name underscore underscore" we'll say "dunder name"

Terminology for Python Special Methods

<method-name>

“dunder”

“double underscore”



Terminology for Python Special Methods

`__len__`

`"dunder len"`

Import or Execute

```
from urllib.request import urlopen

def fetch_words():
    story = urlopen('http://sixty-north.com/c/t.txt')
    story_words = []
    for line in story:
        line_words = line.decode('utf8').split()
        for word in line_words:
            story_words.append(word)
    story.close()

    for word in story_words:
        print(word)

if __name__ == '__main__':
    fetch_words()
```

Docstring

docstrings

Literal strings which document functions, modules, and classes.

They must be the first statement in the blocks for these constructs.

Docstrings

```
>>> from words import *  
>>> help(fetch_words)  
Help on function fetch_words in module words:
```

```
fetch_words(url)
```

```
    Fetch a list of words from a URL.
```

```
    Args:
```

```
        url: The URL of a UTF-8 text document.
```

```
    Returns:
```

```
        A list of strings containing the words from  
        the document.
```

```
(END)
```

Docstrings

Help on module words:

NAME

words - Retrieve and print words from a URL.

DESCRIPTION

Usage:

```
python3 words.py <URL>
```

FUNCTIONS

fetch_words(url)

Fetch a list of words from a URL.

Args:

url: The URL of a UTF-8 text document.

Returns:

A list of strings containing the words from the document.

main(url)

Print each word from a text document from at a URL.

:

Comment

Comments



Code is ideally clear enough without ancillary explanation

Sometimes you need to explain why your code is written as it is

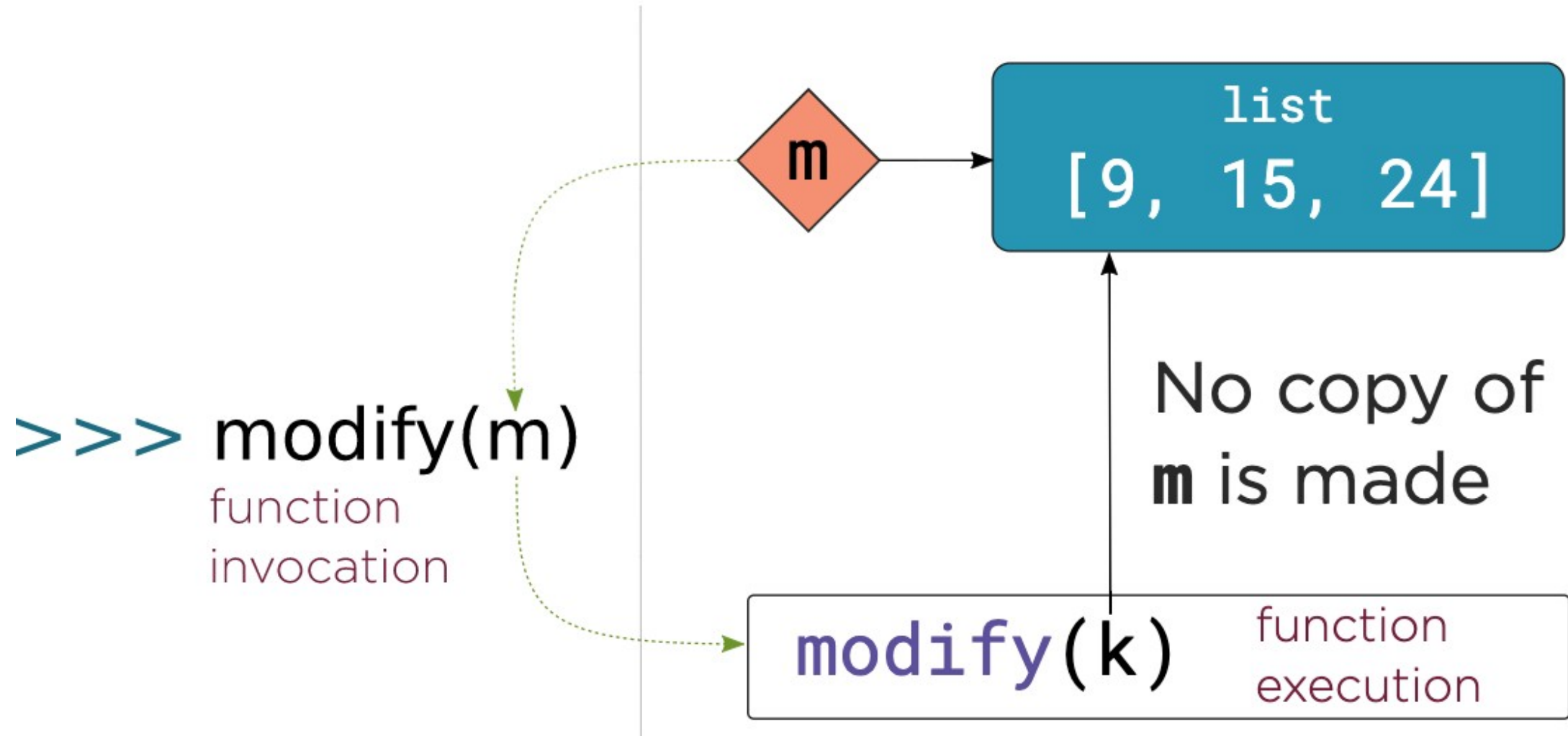
Comments in Python start with # and extend to the end of the line

Passing arguments and returning values

Argument Passing

```
>>> m = [9, 15, 24]
>>> def modify(k):
...     k.append(39)
...     print("k =", k)
...
>>> modify(m)
k = [9, 15, 24, 39]
>>> m
[9, 15, 24, 39]
>>>
```

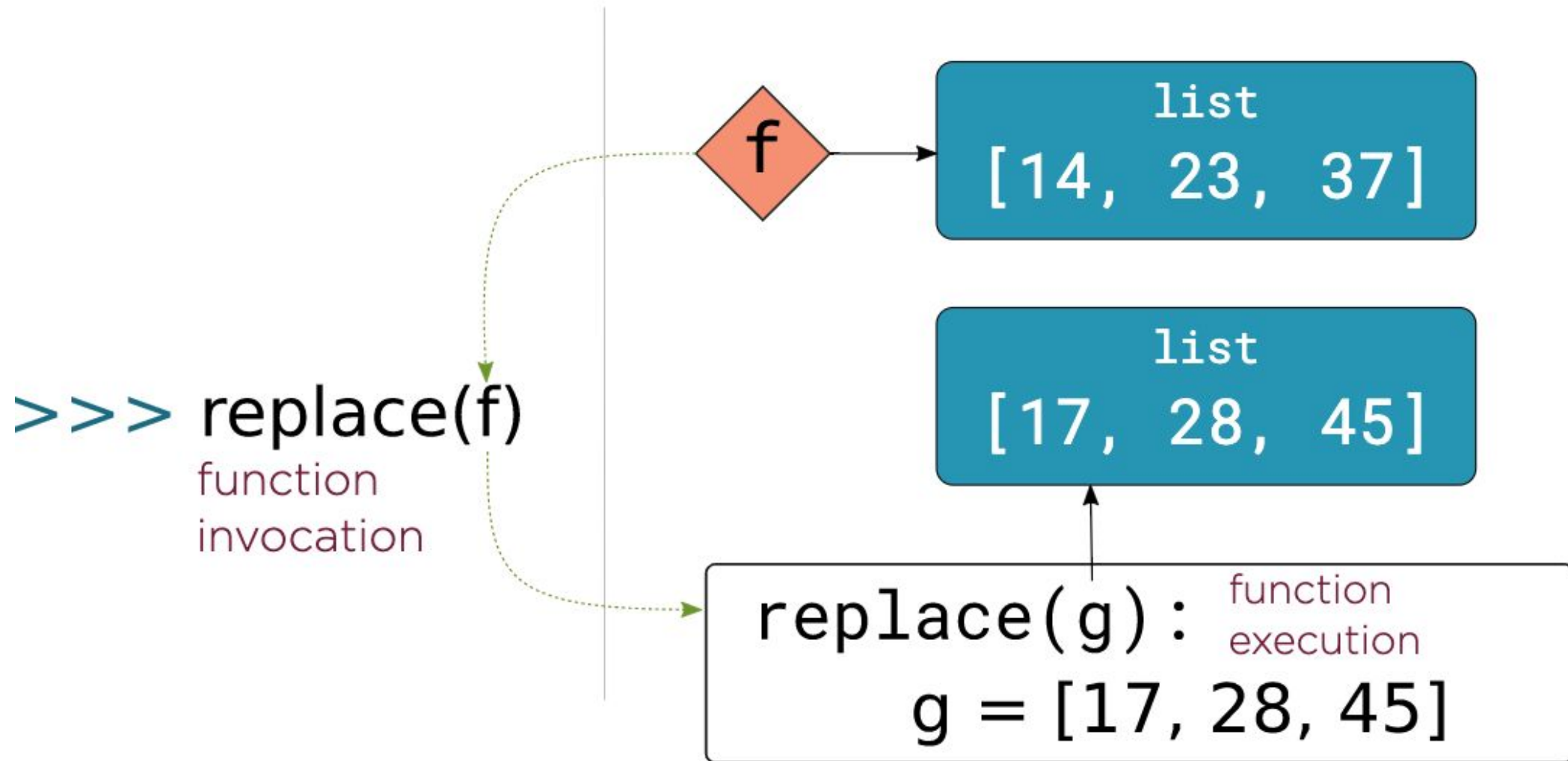
Argument Passing Semantics



Replacing Argument Value

```
>>> f = [14, 23, 37]
>>> def replace(g):
...     g = [17, 28, 45]
...     print("g =", g)
...
>>> replace(f)
g = [17, 28, 45]
>>> f
[14, 23, 37]
>>>
```

Replacing Argument Value



Return Semantics

```
>>> def f(d):  
...     return d  
...  
>>> c = [6, 10, 16]  
>>> e = f(c)  
>>> c is e  
True  
>>>
```

Function Arguments

Default Argument Values

```
>>> def banner(message, border='- '):
...     line = border * len(message)
...     print(line)
...     print(message)
...     print(line)
...
>>> banner("Norwegian Blue")
-----
Norwegian Blue
-----
>>> banner("Sun, Moon and Stars", "*")
*****
Sun, Moon and Stars
*****
>>> banner("Sun, Moon and Stars", border="*")
*****
Sun, Moon and Stars
*****
>>> banner(border=".", message="Hello from Earth")
.....
Hello from Earth
.....
>>>
```


Arguments with default values must come after those without default values.

When are default values evaluated?

Default Value Evaluation

```
>>> import time
>>> time.ctime()
'Sun Nov 24 19:43:48 2019'
>>> def show_default(arg=time.ctime()):
...     print(arg)
...
>>> show_default()
Sun Nov 24 19:43:49 2019
>>> show_default()
Sun Nov 24 19:43:49 2019
>>> show_default()
Sun Nov 24 19:43:49 2019
>>>
```

Default Value Evaluation



Remember that `def` is a statement executed at runtime.

Default arguments are evaluated when `def` is executed.

Immutable default values don't cause problems.

Mutable default values can cause confusing effects.

Always use immutable
objects for default values.

Handle Exception

Exception handling

Mechanism for interrupting normal program flow and continuing in surrounding context

Exceptions: Key Concepts

1. **Raising** an exception
2. **Handling** an exception
3. **Unhandled** exceptions
4. Exception **objects**

Handle Exception

Cleanup Actions

try...finally

try:

try-block

finally:

executed no matter how the

try-block terminates

Not Exception-safe

```
import os
```

```
def make_at(path, dir_name):  
    original_path = os.getcwd()  
    os.chdir(path)  
    os.mkdir(dir_name)  
    os.chdir(original_path)
```

Handle Exception and Cleanup

```
import os
import sys

def make_at(path, dir_name):
    original_path = os.getcwd()
    os.chdir(path)
    try:
        os.mkdir(dir_name)
    except OSError as e:
        print(e, file=sys.stderr)
        raise
    finally:
        os.chdir(original_path)
```

Moment of Zen

**Errors should never
pass silently, unless
explicitly silenced**

Errors are like bells
And if we make them silent
They are of no use



Handle Exception

Exceptions and Control flow

```
DIGIT_MAP = {  
    'zero': '0',  
    'one': '1',  
    'two': '2',  
    'three': '3',  
    'four': '4',  
    'five': '5',  
    'six': '6',  
    'seven': '7',  
    'eight': '8',  
    'nine': '9',  
}
```

```
def convert(s):  
    number = ''  
    for token in s:  
        number += DIGIT_MAP[token]  
    x = int(number)  
    return x
```

◀ **Filename: exceptional.py**

◀ **Define a function**

◀ **Convert string to integer**

◀ **Return the integer**

```
>>> from exceptional import convert
>>> convert("one three three seven".split())
1337
>>> convert("around two grillion".split())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/use-convert-v1/exceptional.py", line 18, in convert
    number += DIGIT_MAP[token]
KeyError: 'around'
>>>
```


Exception Propagation

REPL

convert()

DIGIT_MAP["around"]

KeyError



The diagram illustrates the flow of an exception. It consists of three nested rectangular regions: a dark blue outer region labeled 'REPL', a maroon middle region labeled 'convert()', and a green inner region labeled 'DIGIT_MAP["around"]'. Within the green region is an orange rounded rectangle labeled 'KeyError'. Two orange curved arrows originate from the 'KeyError' box: one points upwards and to the left towards the 'convert()' region, and the other points further upwards and to the left towards the 'REPL' region, representing the propagation of the exception out of the function and into the interpreter.

```
def convert(s):  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
    except KeyError:  
        x = -1  
    return x
```

◀ try-block

◀ Raise exceptions

◀ except-block

◀ Handle exceptions

```
def convert(s):  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
        print(f"Conversion succeeded! x = {x}")  
    except KeyError:  
        print("Conversion failed!")  
        x = -1  
    return x
```

◀ Print on success

◀ Print on failure

```
>>> from exceptional import convert
>>> convert("three four".split())
Conversion succeeded! x = 34
34
>>> convert("eleventeen".split())
Conversion failed!
-1
>>> convert(512)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/use-convert-v3/exceptional.py", line 18, in convert
    for token in s:
TypeError: 'int' object is not iterable
>>>
```

```
def convert(s):  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
        print(f"Conversion succeeded! x = {x}")  
    except KeyError:  
        print("Conversion failed!")  
        x = -1  
    return x
```

◀ Not executed

◀ Executed

```
def convert(s):  
    """Convert a string to an integer."""  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
        print(f"Conversion succeeded! x = {x}")  
    except KeyError:  
        print("Conversion failed!")  
        x = -1  
    except TypeError:  
        print("Conversion failed!")  
        x = -1  
    return x
```

◀ Duplication

◀ Add TypeError handler

◀ Duplication

```
>>> from exceptional import convert
>>> convert(512)
Conversion failed!
-1
>>>
```

```
def convert(s):  
    """Convert a string to an integer."""  
    x = -1  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
        print(f"Conversion succeeded! x = {x}")  
    except KeyError:  
        print("Conversion failed!")  
    except TypeError:  
        print("Conversion failed!")  
    return x
```

◀ Assignment

◀ Duplication

◀ Duplication


```
def convert(s):  
    """Convert a string to an integer."""  
    x = -1  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        x = int(number)  
        print(f"Conversion succeeded! x = {x}")  
    except (KeyError, TypeError):  
        print("Conversion failed!")  
    return x
```

◀ Merge except blocks

```
>>> from exceptional import convert
>>> convert("two nine".split())
Conversion succeeded! x = 29
29
>>> convert("elephant".split())
Conversion failed!
-1
>>> convert(451)
Conversion failed!
-1
>>>
```



Exceptions resulting from **programmer errors**:

IndentationError

SyntaxError

NameError

These should **almost never be caught**.

Handle Exception

Accessing Exception Objects

```
import sys
```

```
DIGIT_MAP = . . .
```

```
def convert(s):
```

```
    try:
```

```
        number = ''
```

```
        for token in s:
```

```
            number += DIGIT_MAP[token]
```

```
        return int(number)
```

```
    except (KeyError, TypeError) as e:
```

```
        print(f"Conversion error: {e!r}",
```

```
              file=sys.stderr)
```

```
    return -1
```

```
>>> from exceptional import convert
>>> convert("fail".split())
Conversion error: KeyError('fail')
-1
>>>
```

Exceptions Can Not Be Ignored

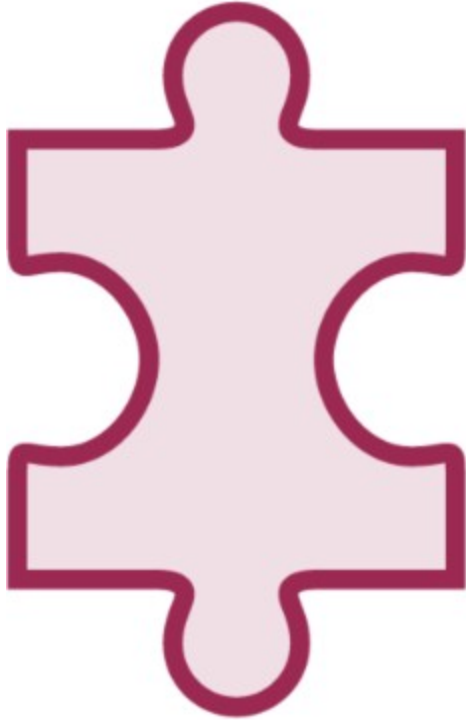


Error codes are **easy to ignore**



Checks are **always required**

Exceptions and Protocols



Sequences should raise `IndexError` for out-of-bounds indexing.

Exceptions must be implemented and documented correctly.

Existing built-in exceptions are often the right ones to use.

IndexError

An integer index is out of range

```
>>> z = [1, 4, 2]
```

```
>>> z[4]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: list index out of range
```

```
>>>
```

ValueError

An object is of the correct type but has an inappropriate value

```
>>> int("jim")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'jim'
>>>
```

KeyError

A lookup in a mapping failed

```
>>> codes = dict.gb=44, us=1, no=47, fr=33, es=34)
>>> codes['de']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'de'
>>>
```

Handle Exception

Avoid explicit type checks

```
def convert(s):  
    if not isinstance(s, list):  
        raise TypeError(  
            "Argument must be a list")  
  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        return int(number)  
    except (KeyError, TypeError) as e:  
        print(f"Conversion error: {e!r}",  
              file=sys.stderr)  
        raise
```

- ◀ Check argument type
- ◀ Raise TypeError


```
def convert(s):  
    # if not isinstance(s, list):  
    #     raise TypeError(  
    #         "Argument must be a list")  
  
    try:  
        number = ''  
        for token in s:  
            number += DIGIT_MAP[token]  
        return int(number)  
    except (KeyError, TypeError) as e:  
        print(f"Conversion error: {e!r}",  
              file=sys.stderr)  
        raise
```

◀ Catch TypeError
◀ Re-raise it