

# Lecture 7

## Topics

- References
- Namespaces
- Exceptions
- Argument Packing
- `argparse`



# Learning Objectives

- You know that binding an existing object to a new variable name does not copy the object.
- You know the (global, local, builtin) python namespaces and how to access them.
- You know how to handle exceptions and how to write custom exceptions.
- You can use the `*` and `**` operators to unpack function arguments.
- You can write command line interfaces (CLIs) using `argparse`.



# References



# New Assignment vs. Object Method



## New assignment

```
In [1]: l = [1, 2, 3]
        c = 1
        l = []
        c
```

```
Out[1]: [1, 2, 3]
```



## Object method

```
In [2]: l = [1, 2, 3]
        c = l
        l.clear()
        c
```

```
Out[2]: []
```



```
In [3]: l = [1, 2, 3]
        c = l
        l[:] = ["a", "b", "c", "d"]
        c
```

```
Out[3]: ['a', 'b', 'c', 'd']
```



# Mutable vs. Immutable Types





# Immutable: String

```
In [4]: s = "Title Case"
        c = s

        s = s.lower()
        s = s.replace("title", "lower")

        s
```

```
Out[4]: 'lower case'
```



# Immutable: String

```
In [4]: s = "Title Case"
        c = s

        s = s.lower()
        s = s.replace("title", "lower")

        s
```

```
Out[4]: 'lower case'
```

```
In [5]: c
```

```
Out[5]: 'Title Case'
```



## Mutable: List

```
In [6]: l = [1, 2, 3]
        c = l

        l.extend([4, 5, 6])
        l[2] = 7

        l
```

```
Out[6]: [1, 2, 7, 4, 5, 6]
```



# Mutable: List

```
In [6]: l = [1, 2, 3]
        c = l

        l.extend([4, 5, 6])
        l[2] = 7

        l
```

```
Out[6]: [1, 2, 7, 4, 5, 6]
```

```
In [7]: c
```

```
Out[7]: [1, 2, 7, 4, 5, 6]
```



# Equality and Identity



```
In [8]: l = ["one", "two", "three"]  
        c = l  
  
        l == c
```

```
Out[8]: True
```



```
In [8]: l = ["one", "two", "three"]  
c = l  
  
l == c
```

Out[8]: True

```
In [9]: l is c
```

Out[9]: True



```
In [10]: l = ["one", "two", "three"]  
c = ["one", "two", "three"]  
  
l == c
```

```
Out[10]: True
```





```
In [10]: l = ["one", "two", "three"]  
c = ["one", "two", "three"]  
  
l == c
```

Out[10]: True

```
In [11]: l is c
```

Out[11]: False



# Loop Variables

```
In [12]: l = [ [1, 2, 3], [4, 5], [6] ]

for elem in l:
    elem.append(0)

print(l)

[[1, 2, 3, 0], [4, 5, 0], [6, 0]]
```



```
In [13]: l = [1, 2, 3]

for elem in l:
    elem += 1

print(l)
print(elem)
```

```
[1, 2, 3]
4
```



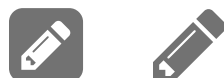
```
In [13]: l = [1, 2, 3]

for elem in l:
    elem += 1

print(l)
print(elem)
```

```
[1, 2, 3]
4
```

- Loop variable references the individual objects over which we iterate
- Immutable objects cannot be changed/replaced like this.



# Classic Mistake

```
In [14]: nested = [[]] * 3  
nested
```

```
Out[14]: [[], [], []]
```

```
In [15]: nested[0].append("first list")  
nested
```

```
Out[15]: [['first list'], ['first list'], ['first list']]
```



```
In [16]: # initialize nested list like this instead:  
nested = [], [], []  
nested[0].append("first list")  
nested
```

```
Out[16]: [['first list'], [], []]
```



# Learning Goals

- `x = y` does not create a copy



# Learning Goals

- `x = y` does not create a copy
- Multiple variables can point to the same changeable object.





# Learning Goals

- `x = y` does not create a copy
- Multiple variables can point to the same changeable object.
- New assignment of a variable does not change its value but binds the name to a different object.



# Namespaces



Definition from the official Python glossary:

***namespace***

*The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces [...].*



# Variables

- Variables in Python are names
- Every variable belongs to exactly one namespace
- It is possible to have variables with the same name in different namespaces



# Variables

- Variables in Python are names
- Every variable belongs to exactly one namespace
- It is possible to have variables with the same name in different namespaces

```
In [17]: open
```

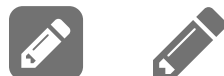
```
Out[17]: <function io.open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)>
```

```
In [18]: import os
os.open
```

```
Out[18]: <function posix.open(path, flags, mode=511, *, dir_fd=None)>
```

```
In [19]: import codecs
codecs.open
```

```
Out[19]: <function codecs.open(filename, mode='r', encoding=None, errors='strict', buffering=-1)>
```



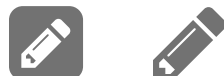
# Name Space as a Dictionary

The namespace of an object is a dictionary.

```
In [20]: class Dish:
          'Create a tasty meal.'
          def __init__(self, spam: int, eggs: int):
              self.spam = spam
              self.eggs = eggs
```

```
In [21]: d = Dish(2, 4)
          d.__dict__
```

```
Out[21]: {'spam': 2, 'eggs': 4}
```



```
In [22]: d.bacon = 3  
         d.__dict__
```

```
Out[22]: {'spam': 2, 'eggs': 4, 'bacon': 3}
```



# Immediate name spaces

Three namespaces\* are directly accessible (without period):

- Local names
- Global names
- `builtins`

When accessing a name, the namespaces are searched in this order.





# Immediate name spaces

Three namespaces\* are directly accessible (without period):

- Local names
- Global names
- `builtins`

When accessing a name, the namespaces are searched in this order.

\* In the case of nested functions, additional namespaces are accessible.



# Immediate name spaces

Three namespaces\* are directly accessible (without period):

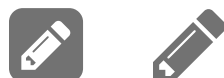
- Local names
- Global names
- `builtins`

When accessing a name, the namespaces are searched in this order.

\* In the case of nested functions, additional namespaces are accessible.

```
In [23]: x = "global namespace"
         x
```

```
Out[23]: 'global namespace'
```



# Immediate name spaces

Three namespaces\* are directly accessible (without period):

- Local names
- Global names
- `builtins`

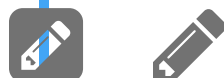
When accessing a name, the namespaces are searched in this order.

\* In the case of nested functions, additional namespaces are accessible.

```
In [23]: x = "global namespace"
         x
```

```
Out[23]: 'global namespace'
```

```
In [24]: def foo():
         x = "local namespace"
         print(x)
```



# Immediate name spaces

Three namespaces\* are directly accessible (without period):

- Local names
- Global names
- `builtins`

When accessing a name, the namespaces are searched in this order.

\* In the case of nested functions, additional namespaces are accessible.

```
In [23]: x = "global namespace"
         x
```

```
Out[23]: 'global namespace'
```

```
In [24]: def foo():
         x = "local namespace"
         print(x)
```



# Function Namespace

- The global namespace is accessible (for reading).
- Assignments always happen in the local function namespace!
- Externally, the local function namespace is not accessible.



# Function Namespace

- The global namespace is accessible (for reading).
- Assignments always happen in the local function namespace!
- Externally, the local function namespace is not accessible.
- Exception: when keywords `global` or `nonlocal` are used.



## global keyword

```
In [27]: mystring = "global string"
```

```
def foo():  
    global mystring  
    mystring = "scope?"
```

```
foo()
```

```
print(mystring)
```

```
scope?
```



## global keyword

```
In [27]: mystring = "global string"

def foo():
    global mystring
    mystring = "scope?"

foo()

print(mystring)

scope?
```

`global` can be used to modify variables from a non-global scope.

Accessing global variables from a non-global scope (for reading) is possible even without `global`.





## nonlocal keyword

```
In [29]: def outer():  
          mystring = "local to outer"  
  
          def inner():  
              nonlocal mystring  
              mystring = "local to inner"  
          inner()  
  
          print(mystring)  
  
outer()
```

local to inner



## nonlocal keyword

```
In [29]: def outer():
          mystring = "local to outer"

          def inner():
              nonlocal mystring
              mystring = "local to inner"
          inner()

          print(mystring)

outer()

local to inner
```

`nonlocal` can be used in nested functions, if the variable should not belong to the scope of the inner function.



# Local Namespace, Scope

Code blocks with their own (local) namespace:

- Modules
- Classes
- Functions/methods



# Local Namespace, Scope

Code blocks with their own (local) namespace:

- Modules
- Classes
- Functions/methods

The part of the code where a name(-space) is directly accessible is called scope.



# Local Namespace, Scope

Code blocks with their own (local) namespace:

- Modules
- Classes
- Functions/methods

The part of the code where a name(-space) is directly accessible is called scope.

Other blocks have no own namespace / scope (`for, while, if, with, try`).



```
import random

AMOUNT_OF_SPAM = 0.3

def spam(text, amount=AMOUNT_OF_SPAM):
    '''Insert the given amount of spam.'''
    spammer = Spammer()
    return spammer.spam(text, amount)

class Spammer:
    '''A collection of spamming utilities.'''

    DEFAULT_TOKEN = 'SPAM!'

    def __init__(self, spamtoken=None):
        self.spamtoken = spamtoken or self.DEFAULT_TOKEN

    def spam(self, text, amount):
        '''Insert spam at random positions.'''
        tokens = text.split()
        spamcount = round(len(tokens)*amount)
        for _ in range(spamcount):
            position = random.randint(0, len(tokens)+1)
            tokens.insert(position, self.spamtoken)
        return ' '.join(tokens)

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



# Exception Handling



# Typical Use Cases





# Logical Structure

Standard case and exception

```
In [32]: import os

try:
    os.mkdir("Lecture Notes")
except FileExistsError:
    print("directory exists, but continue anyway")
```

directory exists, but continue anyway



# Control Structure

Breaking out of an infinite loop

```
try:
    server.serve_forever()
except KeyboardInterrupt:
    clean_up()
    sys.exit(0)
```



# Back-Off

Cascade of trial and error

```
try:
    text = data.decode('utf8')
except UnicodeDecodeError:
    try:
        text = data.decode('cp1252')
    except UnicodeDecodeError:
        text = data.decode('latin1')
```



# Syntax

## Form

```
try:
    [...]
except Exception_1:
    [...]
:
except Exception_n:
    [...]
else:
    [...]
finally:
```



# Execution

- `try` block works:
  - `else` block
  - `finally` block
- try block fails with a planned Exception:
  - first matching except block is executed (only one!)
  - `finally` block
- try block fails with an unplanned Exception:
  - `finally` block
  - Exception is passed on



# Elements

- `except` block
  - handles exceptions
- `else` block
  - **NO** exception in `try` block
  - without protection from new exceptions
- `finally` block
  - executed in all cases
  - ideal for "cleaning up"



# Exception Hierarchy

```
In [35]: try:
        inp = input('Length>> ').split()
        length = float(inp[0])
        unit = inp[-1].encode('ascii')
    except ValueError:
        print('invalid number')
    except UnicodeEncodeError:
        print('only ASCII characters allowed')
```

```
Length>> 10 μm
invalid number
```



# Exception Hierarchy

```
In [35]: try:
inp = input('Length>> ').split()
length = float(inp[0])
unit = inp[-1].encode('ascii')
except ValueError:
    print('invalid number')
except UnicodeEncodeError:
    print('only ASCII characters allowed')
```

```
Length>> 10 μm
invalid number
```

Why?





```
In [36]: UnicodeEncodeError.mro()
```

```
Out[36]: [UnicodeEncodeError,  
          UnicodeError,  
          ValueError,  
          Exception,  
          BaseException,  
          object]
```



```
In [36]: UnicodeEncodeError.mro()
```

```
Out[36]: [UnicodeEncodeError,  
          UnicodeError,  
          ValueError,  
          Exception,  
          BaseException,  
          object]
```

→ `UnicodeEncodeError` is a `ValueError`!

- Exceptions are organised as a class hierarchy
- Exceptions are also caught by their parent class



- Order of except statements is relevant: More specific cases first

```
In [37]: try:
inp = input('Length>> ').split()
length = float(inp[0])
unit = inp[-1].encode('ascii')
except UnicodeEncodeError:
    print('only ASCII characters allowed')
except ValueError:
    print('invalid number')
```

```
Length>> 10 μm
only ASCII characters allowed
```



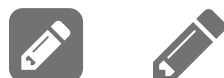
# Inspection of an Exception

Caught exceptions can be bound to a variable with `as`

Access to its arguments is granted with `.args` (→ Tuple)

```
In [38]: d = dict(tokens=20, types=12)
try:
    print(d["lines"])
except KeyError as e:
    # dict raises a key error if key does not exist.
    print(e.args)
    print(f"unknown key: {e.args[0]}")
```

```
('lines',)
unknown key: lines
```

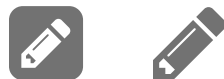


Some exceptions have more information:

In [39]:

```
try:
    'Računalnik'.encode('ascii')
except UnicodeEncodeError as e:
    print(e.args)
```

```
('ascii', 'Računalnik', 2, 3, 'ordinal not in range(128)')
```



# Passing Exceptions On

## Create an exception

```
In [40]: token = 123

if not isinstance(token, str):
    raise TypeError('expected str, got {}'.format(type(token)))
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[40], line 4
      1 token = 123
      3 if not isinstance(token, str):
----> 4     raise TypeError('expected str, got {}'.format(type(token)))

TypeError: expected str, got <class 'int'>
```



# Changing the error type

```
In [41]: def average(seq):  
        try:  
            return sum(seq) / len(seq)  
        except ZeroDivisionError:  
            raise ValueError('sequence must not be empty')
```

```
In [43]: average([])
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
Cell In[41], line 3, in average(seq)  
      2 try:  
----> 3     return sum(seq) / len(seq)  
      4 except ZeroDivisionError:
```

ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

```
ValueError                                Traceback (most recent call last)  
Cell In[43], line 1  
----> 1 average([])
```

```
Cell In[41], line 5, in average(seq)  
      3     return sum(seq) / len(seq)  
      4 except ZeroDivisionError:  
----> 5     raise ValueError('sequence must not be empty')
```

ValueError: sequence must not be empty



## Defining your own exception classes

- Use built-in classes where they make sense (e.g. `ValueError`, `TypeError`, `KeyError`)
- If needed, write own exceptions

```
class InvalidFormatError(Exception):  
    '''Input data does not conform to the CoNLL format.'''
```

Important: Exceptions should always inherit from `Exception` (or one of its subclass)!





- Exceptions with arguments:

```
In [44]: class InvalidFormatError(Exception):
          '''Input data does not conform to the CoNLL format.'''

          def foo(fields):
              if len(fields) < 3:
                  msg = f'too few columns: expected 3, got {len(fields)}'
                  raise InvalidFormatError(msg)
```

```
In [45]: foo(['col1', 'col2'])
```

```
-----
InvalidFormatError                                Traceback (most recent call last)
Cell In[45], line 1
----> 1 foo(['col1', 'col2'])

Cell In[44], line 7, in foo(fields)
      5 if len(fields) < 3:
      6     msg = f'too few columns: expected 3, got {len(fields)}'
----> 7     raise InvalidFormatError(msg)

InvalidFormatError: too few columns: expected 3, got 2
```



# Bad Habits

```
try:  
    [many  
    lines  
    of  
    code]  
except:  
    ...
```



# Bad Habits

```
try:  
    [many  
    lines  
    of  
    code]  
except:  
    ...
```

`try` blocks should be short



```
try:  
    # do something  
except:  
    pass
```



```
try:
    # do something
except:
    pass
```

`except` without a type catches everything (including e.g. `KeyboardInterrupt`)



```
try:  
    # do something  
except Exception as e:  
    print(e)
```



```
try:
    # do something
except Exception as e:
    print(e)
```

Handle errors or pass them on with `raise`, but do not just continue



# Klicker Quiz

<https://pwa.klicker.uzh.ch/join/lfische>





# Argument Packing



# Variable number of arguments

```
In [46]: max('123')
```

```
Out[46]: '3'
```

```
In [47]: max('123', '456')
```

```
Out[47]: '456'
```



# Variable number of arguments

```
In [46]: max('123')
```

```
Out[46]: '3'
```

```
In [47]: max('123', '456')
```

```
Out[47]: '456'
```

How can we write such a function?



```
In [51]: from typing import Iterable, Any
def find_largest(sequence: Iterable) -> Any:
    "find the largest element in a sequence"
    largest = sequence[0]
    for element in sequence[1:]:
        if element > largest:
            largest = element
    return largest
```



```
In [48]: def custom_max(*args):  
         # store all arguments as a tuple in variable args  
         if len(args) == 0:  
             raise TypeError("max expects at least one argument")  
         if len(args) == 1:  
             return find_largest(args[0])  
         else:  
             return find_largest(args)
```



```
In [48]: def custom_max(*args):  
        # store all arguments as a tuple in variable args  
        if len(args) == 0:  
            raise TypeError("max expects at least one argument")  
        if len(args) == 1:  
            return find_largest(args[0])  
        else:  
            return find_largest(args)
```

```
In [53]: custom_max('123', '4634', '2343')
```

```
Out[53]: '4634'
```



# Positional and Keyword Arguments

## Positional

\*args → tuple

```
In [55]: def func(*args):  
          print(args)  
  
          func(1, 2, "three", (1,2))  
  
          (1, 2, 'three', (1, 2))
```



Unpack any iterable with `*` into function arguments:

```
In [56]: params = "Hello"  
func(*params)  
  
('H', 'e', 'l', 'l', 'o')
```





# Keyword arguments

`**kwargs` → dict

```
In [57]: def func(**kwargs):  
          print(kwargs)  
  
          func(a=1, b='B')  
  
          {'a': 1, 'b': 'B'}
```



Unpack dictionaries with `**` into function arguments:

```
In [58]: d = dict(city="Zürich", postal_code=8050)
func(**d)

{'city': 'Zürich', 'postal_code': 8050}
```



# Mixed Arguments

```
In [59]: def func(a, b, *args, x=3, y=5, **kwargs):  
         print(f'a: {a}, b: {b}, args: {args}, x: {x}, y: {y}, kwargs: {kwargs}')
```

```
In [63]: func(0, x=1, a=2)
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[63], line 1  
----> 1 func(0, x=1, a=2)  
  
TypeError: func() got multiple values for argument 'a'
```



# Mixed Arguments

```
In [59]: def func(a, b, *args, x=3, y=5, **kwargs):  
         print(f'a: {a}, b: {b}, args: {args}, x: {x}, y: {y}, kwargs: {kwargs}')
```

```
In [63]: func(0, x=1, a=2)
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[63], line 1  
----> 1 func(0, x=1, a=2)  
  
TypeError: func() got multiple values for argument 'a'
```

```
In [71]: params = {'a': 0, 'b': 1, 'c': 2}  
         func(*params)
```

```
a: a, b: b, args: (), x: c, y: 5, kwargs: {}
```



Alternative order:

```
In [65]: def func(a, b, x=3, y=5, *args, **kwargs):  
         print(f'a: {a}, b: {b}, args: {args}, x: {x}, y: {y}, kwargs: {kwargs}')
```

```
In [66]: func(0, 1, 2, c=3, y=7)
```

```
a: 0, b: 1, args: (), x: 2, y: 7, kwargs: {'c': 3}
```



## Exercise: Practice Function Calls With Argument Packing

```
def func(a, b, x=3, y=5, *args, **kwargs):  
    ...
```

Try these function calls. Do they work?

```
params = [0, 1]  
func(*params)
```

```
params = (3, 2)  
func(*params, y=1, b=2)
```

```
params = {'a': 0, 'b': 3, 'c': 2}  
func(*params, y=1)
```



```
In [67]: params = [0, 1]
func(*params)
```

```
a: 0, b: 1, args: (), x: 3, y: 5, kwargs: {}
```

```
In [68]: params = (3, 2)
func(*params, y=1, b=2)
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[68], line 2
      1 params = (3, 2)
----> 2 func(*params, y=1, b=2)
```

```
TypeError: func() got multiple values for argument 'b'
```

```
In [70]: params = {'a': 0, 'b': 3, 'c': 2}
func(**params, y=1)
```

```
a: 0, b: 3, args: (), x: 3, y: 1, kwargs: {'c': 2}
```



**argparse**





# Motivation

A command line interface (CLI) provides a way for a user to interact with a program running in a text-based shell interpreter.

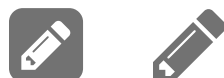


```
import sys
if __name__ == "__main__":
    for i, arg in enumerate(sys.argv):
        print(f"Argument {i}: {arg}")
```

---

```
$ python main.py arg1 arg2 arg3 arg4
```

With `sys`, you need to remember how many / what arguments your program takes and which argument should be provided at what position.



## The Module **argparse** ...

- automatically generates help messages
- provides detailed information about what each argument should be
- greatly increases user experience



Important calls:

```
parser = ArgumentParser() # Creates a parser object
```

```
parser.add_argument() # Defines how an argument should be parsed
```

```
parser.parse_args() # Parses arguments from command line with correct type
```



# argparse Arguments

## Optional Arguments

```
parser.add_argument( '-n', '--number', type=int, help="a number" )
```



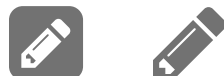
# argparse Arguments

## Optional Arguments

```
parser.add_argument('-n', '--number', type=int, help="a number")
```

## Required Arguments

```
parser.add_argument('-n', '--number', type=int, help="a number", required=True)
```



## Positional Arguments

```
parser.add_argument('n1', type=int, help="first number")  
parser.add_argument('n2', type=int, help="second number")
```



## Positional Arguments

```
parser.add_argument('n1', type=int, help="first number")  
parser.add_argument('n2', type=int, help="second number")
```

## Choice Arguments

```
parser.add_argument('n', type=int, choices=[0, 1, 2], help="either 0, 1 or 2")
```





## Default Arguments

```
parser.add_argument('-n', '--number', type=int, default=0, help="a number")
```

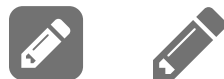
## Append Action

*# Appends all values with '-n' to a list*

```
parser.add_argument('-n', type=int, action='append', help="a list of numbers")
```

---

```
$ python test.py -n 1 -n 2 -n 3
```



## Multiple Arguments

```
# Similar behaviour as append action, given as -n1 1 2 3  
parser.add_argument('n1', type=int, nargs='3', help="a list of three numbers")  
parser.add_argument('n2', type=int, nargs='+', help="a list of 1 ... n numbers")  
parser.add_argument('n3', type=int, nargs='*', help="a list of 0 ... n numbers")
```

---

```
$ python test.py -n1 1 2 3 -n2 4 5 6 7 8
```



# argparse Demo



# Take-home messages

- Binding an existing object to a new variable name does not copy the object.
- Python namespaces (global, local, builtin) store variables and objects.
- `try/except` blocks are used to handle exceptions, and `raise` is used to throw exceptions.
- The `*` and `**` operators are used to (un)pack function arguments.
- Use `argparse` to write command line interfaces (CLIs).

