07_Namespaces_References

April 9, 2024

Programming Techniques in Computational Linguistics II – FS23

1 Lecture 7

1.1 Topics

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

1.2 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`.

2 References

2.1 New Assignment vs. Object Method

2.1.1 New assignment

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

2.1.2 Object method

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

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

`l[:]` calls `__setitem__` on the list

2.2 Mutable vs. Immutable Types

2.2.1 Immutable: String

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

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

s
```

[ ]:
```
c
```

2.2.2 Mutable: List

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

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

l
```

[ ]:
```
c
```

2.3 Equality and Identity

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

l == c
```

[ ]:
```
l is c
```

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

l == c
```

[ ]:
```
l is c
```

2.4 Loop Variables

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

for elem in l:
    elem.append(0)

print(l)
```

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

for elem in l:
    elem += 1

print(l)
print(elem)
```

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

2.5 Classic Mistake

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

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

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

2.6 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.

3 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 [...].

3.1 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

[ ]:
```
open
```

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

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

3.2 Name Space as a Dictionary

The namespace of an object is a dictionary.

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

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

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

3.3 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.

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

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

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

[ ]:
```
print(x)
foo()
print(x)
```

3.4 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.

3.4.1 `global` keyword

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

def foo():
    mystring = "scope?"

foo()

print(mystring)
```

`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`.

3.4.2 `nonlocal` keyword

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

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

    print(mystring)

outer()
```

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

3.5 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`).

4 Exception Handling

4.1 Typical Use Cases

4.1.1 Logical Structure

Standard case and exception

[ ]:
```
import os


os.mkdir("Lecture Notes")
```

4.1.2 Control Structure

Breaking out of an infinite loop

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

4.1.3 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')
```

4.2 Syntax

4.2.1 Form

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

```
    [...]
finally:
    [...]
```

4.2.2 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

4.2.3 Elements

- `except` block
  - handles exceptions
- `else` block
  - **NO** exception in `try` block
  - whitout protection from new exceptions
- `finally` block
  - executed in all cases
  - ideal for "cleaning up"

4.3 Exception Hierarchy

[ ]:
```
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>> two mm
invalid number

Length>> 34  m
invalid number
```

Why?

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

→ `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

[ ]:
```
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')
```

4.4 Inspection of an Exception

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

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

[ ]:
```
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]}")
```

Some exceptions have more information:

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

4.5 Passing Exceptions On

4.5.1 Create an exception

[ ]:
```
token = 123

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

4.5.2 Changing the error type

[ ]:
```
def average(seq):
    try:
        return sum(seq) / len(seq)
    except ZeroDivisionError:
```

```
        raise ValueError('sequence must not be empty')
```

[ ]:
```
average([])
```

4.5.3 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:

[ ]:
```
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)
```

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

4.6 Bad Habits

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

`try` blocks should be short

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

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

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

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

4.6.1 Klicker Quiz

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

5 Argument Packing

5.1 Variable number of arguments

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

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

How can we write such a function?

[ ]:
```
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
```

[ ]:
```
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)
```

[ ]:
```
custom_max()
```

5.2 Positional and Keyword Arguments

5.2.1 Positional

*args → `tuple`

[ ]:
```
def func(*args):
    print(args)

func(1, 2, 3)
```

Unpack any iterable with * into function arguments:

[ ]:
```
params = "Hello"
func(*params)
```

5.2.2 Keyword arguments

**kwargs → `dict`

```
[ ]: def func(**kwargs):
         print(kwargs)

     func(a=1, b='B')
```

Unpack dictionaries with ** into function arguments:

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

5.2.3 Mixed Arguments

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

```
[ ]: func(0, 1, 2, 3)
```

```
[ ]: func(0, x=1, b=2)
```

```
[ ]: func(0, x=1, y=2)
```

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

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

Alternative order:

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

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

5.2.4 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)
```

[ ]:
```
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)
```

6 argparse

6.1 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.

6.1.1 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
```

6.2 `argparse` Arguments

6.2.1 Optional Arguments

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

6.2.2 Required Arguments

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

6.2.3 Positional Arguments

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

6.2.4 Choice Arguments

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

6.2.5 Default Arguments

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

6.2.6 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
```

6.2.7 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
```

6.3 `argparse` Demo

6.4 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).