aqtivate-lecture

March 27, 2025

1 Python in the context of different programming languages paradigms

- Python is a high-level language, implemented in C through CPython
- Python is an **interpreted** language (C/C++/Julia are compiled)
- Python is dynamically typed, but type-annotations are available (C/C++/Java are statically typed)

- Python offers paradigms such as object-oriented programming (Java, C++) and functional programming (Haskell)
- Python uses mandatory indentation levels to identify code blocks (C/C++/Java use `{}`)
- Python offers a vast standard library (similar to Java/C++, in contrast to C or Lua)
- Python is slow, much slower than C/C++ or even Java, but allows simple cross-language binding with C (and others)

2 Types

Everything in Python are objects, and every object has a type

[ ]:
```python
var = True  # variable
# `print` and `type` are functions
print(type(var))
```

[ ]:
```python
# `print` can be omitted for the code in a cell's last line
# Output is a tuple datatype of two elements
type(1), type(1.0)
```

[ ]:
```python
var = type(1), type(1.0)
type(var)
```

3 Operators on simple types

Operators describe operations between objects. Python code is typed dynamically.

[ ]:
```python
1 + 3 - 2 * 3 / 4.0
```

[ ]:
```python
1 == 1.0
```

[ ]:
```python
a = True
not a
```

[ ]:
```python
a = True
# standard boolean comparison operators
a and a or not a
```

[ ]:
```python
# Type conversions
var = 1
type(var), type(float(var))
```

4 Datastructures

[ ]:
```python
# strings are immutable
# strings can use `''` or `""` literals (be consistent)
a = 'hello'
type(a)
```

[ ]:
```python
# lists are mutable
a = [1, 2, 3]
type(a)  # mutable
```

[ ]:
```python
# tuples are immutable
a = (1, 2, 3)
type(a) # immutable
```

[ ]:
```python
# sets are immutable and encapsulate unique elements
a = {1, 2, 3, 3}
a, type(a)
```

[ ]:
```python
# sets are unordered
a = {3, 3, 2, 1}
a
```

[ ]:
```python
# Dictionary
# keys are immutable (e.g., a tuple can be key)
# O(1) lookup complexity
a = {'a': 1, 'b': 2}
type(a)
```

5 Basic operators on datastructures

[ ]:
```python
"Hel" + "lo"
```

[ ]:
```python
[1, 2, 3] + [5, 4]
```

[ ]:

```
[1, 2] * 2
```

[ ]:

```
# union set
{1, 2, 3} | {2, 3, 4}
```

[ ]:

```
# Intersection of two sets
{1, 2, 3} & {2, 3, 4}
```

[ ]:

```
d = {'a': 1, 'b': 2}
d.update({'c': 3})
print(d)
```

[ ]:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a is b, a is a)
print(a == b)
```

[ ]:

```
# membership operators
'ell' in 'hello', 'hi' in 'hello'
```

[ ]:

```
# container types check for equality of any element
print([2, 3] in [[2], 3, 4, [2,3]])
```

[ ]:

```
# string formatting
day = 13
suffix = 'th'
day_of_week = 'Tuesday'
month = 'February'

f'Today is {day_of_week}, {day}{suffix} of {month}'
```

[ ]:

```
# lazy evaluation (e.g., logging)
print('The weight of %d %s is approximately %.1f g' % (3, 'Apples', 85 * 3))
```

6 Indexing

[ ]:

```
# Indexing of lists, tuples, strings, etc. is similar
[1, 2, 3, 4, 5][1], (1, 2, 3, 4, 5)[1]
```

[ ]:

```
val = [1,2,3]
val[2] = 1
print(val)
```

[ ]:

```
val = (1,2,3)
try:
    val[2] = 1
```

```
# except Exception:
#     pass
    # try:
    #     print(e)
    # except NameError:
    #     pass
except TypeError as e:
    print(e)
```

[ ]:

```
[1, 2, 3, 4, 5][-2]
```

[ ]:

```
[1, 2, 3, 4, 5][:2] # slicing
```

[ ]:

```
[1, 2, 3, 4, 5][:-1]
```

[ ]:

```
[1, 2, 3, 4, 5][-1:]
```

[ ]:

```
[1, 2, 3, 4, 5][::-1]  # reverting
```

[ ]:

```
a = {'a': 1, 'b': 2}
a['b'] = 3
a
```

[ ]:

```
# unpacking
(*(1, 2),3)
```

[ ]:

```
# Dictionaries are ordered by insertion time
dict_a = {'a': 1, 'b': 2, 'c': 3}
dict_b = {'b': 2, 'a': 1, 'c': 3}
print(dict_a, dict_b)

dict_b['a'] = 0  # Modifying an existing memory location
print(dict_b)
```

[ ]:

```
# Unpack dictionaries using the `**` operator (analogous to sequence unpacking)
{**{'a': 2, 'b': 3}, 'c':3}
```

7 Flow control and iteration

[ ]:

```
# tuple unpacks automatically
a, b, c = 2, 3, 4

if a == 2:
    print('a')
elif a > 2:
    print('b')
else:
    print('c')
```

[ ]:
```
for val in [3, 1, 2]:
    print(val)
```

[ ]:
```
for val in [3, 1, 2, 4]:
    if val == 2:
        break
    print(val)
```

[ ]:
```
for val in [3, 1, 2, 4]:
    if val == 2:
        continue
    print(val)
```

[ ]:
```
# lazy evaluation
for val in range(2, 5):
    print(val)
```

[ ]:
```
for k, v in dict_a.items():
    print(k, v)
```

[ ]:
```
names = ['Anna', 'Bob', 'Carl']
ages = [21, 18, 34]
list(zip(names, ages))
```

In the code cell below, notice that sets are unordered. Therefore, we cannot reliably predict the way they will be unpacked by the `zip` function. Of order matters, it is best to convert the set into ordered datastructure like a tuple (if it should be immutable) or a list (if it should be mutable).

[ ]:
```
names_and_ages = zip(names, ages, (0, 1, 2), {1, 32, 4})
for el in names_and_ages:
    print(el)
type(names_and_ages), list(names_and_ages)
```

As a workaround, we could try to guess the internal datastructure of the set. However, note that the resulting code is unrobust and only for educational purposes:

[ ]:
```
s = {1, 32, 4}
print([v for v in s])  # get an idea of how the set could be stored internally
print(list(zip(s, s)))  # the ordering ideally matches the one from the␣
 ↪previous line

# Notice that the order may diverge based on the method used to produce the␣
 ↪standard output (more on the next slide)
print(f'Order from the print function: {s}')
s
```

The `print` function points to the `__str__` dunder method. Evaluating an expression directly in an interactive environment points to the `__repr__` method. More on object-oriented programming later.

[ ]:

```
class Test:
    def __str__(self):
        return 'using str'

    def __repr__(self):
        return 'using repr'

test = Test()
print(test)
test
```

[ ]:

```
for n, tag in enumerate({'yes', 'no', 'maybe'}):
    print(f'{n} -> {tag}')
```

8 List iteration/comprehensions

[ ]:

```
lc = [i for i in range(10)]
lc
```

[ ]:

```
# generator expression
ge = (i for i in range(10))
ge, type(ge)
```

[ ]:

```
import sys
sys.getsizeof(lc), sys.getsizeof(ge)
```

[ ]:

```
[i for i in range(10) if i % 2 == 0]
```

[ ]:

```
# ternary condition operator
[i if i % 2 == 0 else -i for i in range(10)]
```

[ ]:

```
names = ['Anna', 'Bob', 'Carl']
ages = [21, 18, 34]
dct = {name: age for name, age in zip(names, ages)}
dct = {age: name for name, age in zip(names, ages)}
type(dct), dct

dict(zip(ages, names))
```

9 String utility functions

[ ]:

```
'   the story \r\n\r\n '.strip()  # strip leading and trailing whitespaces/
 ↪newlines
```

[ ]:

```
# find index of a sub-string
txt = 'Hello world !'
first = txt.find('wo')
first, txt[first]
```

[ ]:

```
'Apples,Lettuce,Bread'.split(',')  # split a string into a list of strings␣
 ↪given a delimiter
```

[ ]:

```
' and '.join(['Huey', 'Dewey', 'Louie'])  # joining a list of strings using a␣
 ↪delimiter
```

10 Functions

[ ]:

```
# Keyword arg and docstring are optional
# Function arguments specified in the function signature should be immutable
def fun(a, b=1):
    """Adds to numbers

    Args:
        a: the first number
        b: the second number

    Returns:
        the sum of a and b
    """
    return a + b
print(fun(2))
print(fun(2, 3))
```

[ ]:

```
def fun(a: int, b: int = 1) -> int:
    assert isinstance(a, int)
    assert isinstance(b, int)
    return a + b
print(fun(2))
```

[ ]:

```
# optional number of arguments (args) and keyword arguments (kwargs)
def fun(*args, **kwargs):
    return fun2(*args, **kwargs)

def fun2(a, b=1, c=2):
    return a, b, c
```

```
print(fun(1))
print(fun(1, 2))
print(fun(1, c=1, b=2))
try:
    print(fun())
except TypeError as e:
    print(e)
```

[ ]:

```
def fun(list_):
    list_[0] = 4

a = [1, 2, 3]  # mutable
fun(a)
a
```

[ ]:

```
def fun(a_):
    a_ = 4

a = 2  # immutable
fun(a)
a
```

[ ]:

```
# Functions behave like any other variable
def fun(a):
    return a + 2

my_function = fun
my_function(2)
```

[ ]:

```
def square_fn(x: float):
    return x ** 2

def do_twice(func, x):
    return func(func(x))

do_twice(square_fn, 2)
```

11 Object-oriented programming

- Class names are CamelCase by convention
- Methods are functions bound to objects.
- Class methods are called using the syntax .

[ ]:

```
class Student:
    def test():
        print('accessing the test method')
```

```
Student.test()
```

- Methods with two leading and trailing underscores are special/dunder methods.
- Constructors instantiate instances of a class

[ ]:

```
class Student:
    def __init__(self, name=None):  # Constructor
        # Instance attributes
        # `self` points to the instance (name by convention)
        self.name = name

# The below does not work, as we need to (implicitly) pass a reference to the␣
 ↪instance.
try:
    Student.__init__('Marc')
except AttributeError as e:
    print(e)

student = Student('Marc')  # This automatically takes care of the above problem
student.__init__('Ana')  # We can now also re-instantiate manually
print(student.name)
```

Class inheritance: - Place parentclass in parantheses after class name - `super()` points to parent class

[ ]:

```
class StudentAssistant(Student):
    def __init__(self, name=None, employer=None):
        super().__init__(name)
        self.employer = employer
```

[ ]:

```
# Create an instance of the Student class
student = Student(name='John')
# Create an instance of the StudentAssistant class, which inherits from student
studa = StudentAssistant(name='Anna', employer='TUB')
print(studa.name)
```

[ ]:

```
type(student), type(studa)
```

[ ]:

```
isinstance(studa, Student)
```

[ ]:

```
class StudentAssistant(Student):
    def __init__(self, name=None, employer=None):
        super().__init__(name)
        self.employer = employer

    def is_employed_at_tub(self):
        return self.employer == 'TUB'
```

```python
studa = StudentAssistant(name='Anna', employer='TUB')
print(studa.is_employed_at_tub())
```

[ ]:

```python
# You can try uncommenting the `__str__` or `__repr__` methods in the code below.

class StudentAssistant(Student):
    def __init__(self, name=None, employer=None):
        super().__init__(name)
        self.employer = employer

    def is_employed_at_tub(self):
        return self.employer == 'TUB'

    # def __str__(self): # "for users", i.e., the output should be easily understandable
    #     return f'Name: {self.name}'

    def __repr__(self): # "for developers"
        return str(hash(f'Name: {self.name}'))

studa = StudentAssistant(name='Anna', employer='TUB')
```

[ ]:

```python
print(studa, studa.__str__(), studa.__repr__())
```

[ ]:

```python
class StudentA:
    pass

class StudentB(object):
    pass

print(object, type(object))
```

[ ]:

```python
# function `dir` returns attributes and methods
dir(StudentA)
```

[ ]:

```python
delta = set(dir(StudentA)) - set(dir(object))
print(delta)
```

[ ]:

```python
# function `getattr` returns an attribute's value
print(
    [(v, getattr(StudentA(), v)) for v in delta]
)
```

12 Reading data from a file

Content of file *scores.txt* that lists the performance of players at a certain game:

```
80,55,16,26,37,62,49,13,28,56

43,45,47,63,43,65,10,52,30,18

63,71,69,24,54,29,79,83,38,56

46,42,39,14,47,40,72,43,57,47

61,49,65,31,79,62,9,90,65,44

10,28,16,6,61,72,78,55,54,48
```

[ ]:

```python
# the `with` statement openes and closes the file
# `f` is then only available within the code block
with open('./scores.txt', 'r') as fd:
    data = []
    for line in fd:
        line_entries = line.strip().split(',')
        lst = [float(x) for x in line_entries]
        # data.append(lst)
        data.extend(lst)

print(f'Data length: {len(data)}')
print(f'File content: {data}')
```

12.1 Python debugger

- Import
  - Standard library: `import pdb`
  - Jupyter notebooks: `from IPython.core.debugger import set_trace`
- Use the `set_trace()` function to set a breakpoint (the code stops here and an interactive console opens)
- Use `continue`, `next`, or `quit` (or many additional commands) to navigate in the respective interactive shell.
- Some IDEs contain much more sophisticated debuggers (e.g., PyCharm)

[ ]:

```python
from IPython.core.debugger import set_trace

def func():
    a = [5, 6, 7]
    # Uncomment the below so that an interactive shell opens, where you can␣
 ↪evaluate variables
    set_trace()
    b = ['a', 'b']
    pass

func()
```