Symbolische Programmiersprache - Lecture 2

Prof. Dr. Barbara Plank

Symbolische Programmiersprache
MaiNLP, CIS LMU Munich

WS2025/2026

Outline

Python fundamentals

- Mutable versus immutable data types
- Lists

Mutable vs. immutable types

mutable = veränderbar; immutable = unveränderbar

Mutable	**Immutable**
list	integer, float, string, boolean, ...
can change (elements can be added, deleted, modified)	never change; always new references are created

What does it mean to be immutable for a string? Example:

```
x = 'a'
x = 'b'
```

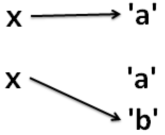
Mutable vs. immutable types

mutable = veränderbar; immutable = unveränderbar

Mutable	**Immutable**
list	integer, float, string, boolean, ...
can change (elements can be added, deleted, modified)	never change; always new references are created

What does it mean to be immutable for a string? Example:

```
x = 'a'
x = 'b'
# the variable x holding string 'a'
# is not changed, a new object with
# the string value 'b' is created!
```

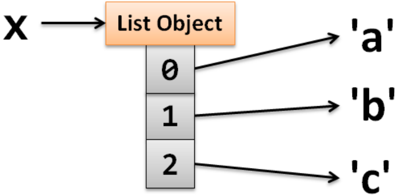


Lists

```
x = ['a', 'b', 'c']
```

Lists

```
x = ['a', 'b', 'c']
```

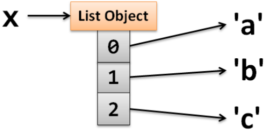


List: Methods

- Example to modify the list object: **append()**

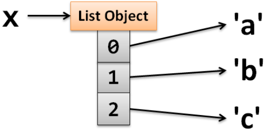
Mutable vs. immutable types

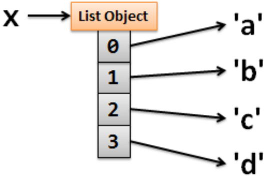
```
>>> x = ['a', 'b', 'c']
>>> x.append('d') # What happens
```



Mutable vs. immutable types

```
>>> x = ['a', 'b', 'c']
>>> x.append('d') # What happens

>>> print(x)
['a', 'b', 'c','d']
```



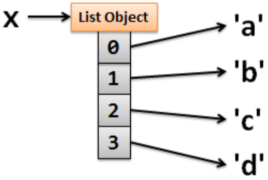



```
# x references the same
# list object x (list=mutable), which is now modified
```

Mutable vs. immutable types

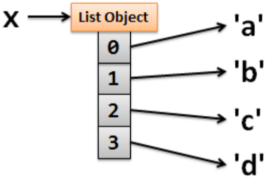
Mutable	Immutable
list	integer, float, string, boolean, ...

```
x = ['a', 'b', 'c']
x.append('d')
x[2] = 'e' # What happens now?
```

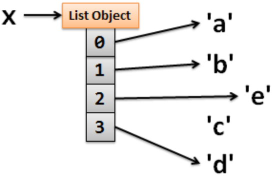


Mutable	Immutable
list	integer, float, string, boolean, ...

```
x = ['a', 'b', 'c']
x.append('d')
x[2] = 'e' # What happens now?
```




```
# x references a list object
# which is modified
# The string object 'c'
# is not modified,
# 'e' is a new string object referenced to
```



Object-Oriented Programming (OOP)

Object Oriented Programming is a way of computer programming which uses the idea of **"objects"** to represents data and methods. It creates reusable code instead of redundant one.

Overview: Procedural/Imperative vs OOP

Procedural	Object Oriented
Program divided using functions	Program divided using objects
Does not make use of access modifiers	Encapsulation: Allows access modifiers (private, public, etc)
Does not support polymorphism	Supports polymorphism
Does not support inheritance	Supports inheritance

Imperative/Procedural Programming

Imperative / Procedural Programming

Imperative Paradigm

First do this, then do that.

- Latin *imperare* for *to command, to order* ⇒ command as a central construction
- **Control Structures** define the order in which the programming steps are executed
- **Data Structures** define how the data is organized
- **Functions and Processes** define the structure ⇒ procedural programming
- State of the program changes as a function of time

Imperative / Procedural Programming

Examples of Procedural Programming Languages
Pascal, Fortran, Algol, C, Lisp

Advantages
Easy to understand and follow due to step-wise execution of commands

Disadvantages

- sequential execution: no easy implementation of shared execution on multiple processors
- side-effects: methods can change the state of the program unexpectedly

Note: for small tasks (e.g. pre-processing scripts) a procedural programming style can be preferred over OOP (e.g. a Python script)

Python scope

Python Scope of a Variable

- Before we move to OOP, an important note on scope in Python
- A variable is only available from inside the region it is created. This is called scope.
- Python distinguishes different levels of scope. The most basic one is:
  - **Local Scope:** a variable created inside a function belongs to the local scope of that function, and can only be used inside that function.
- See `https://www.w3schools.com/python/python_scope.asp` and `https://www.datacamp.com/tutorial/scope-of-variables-python`

Local scope

```
1 def myfunc():
2   # local scope
3   x = 300
4   print(x)
5
6 myfunc()
```

- Whenever you define a variable within a function, its scope lies only within the function. It is accessible from the moment it is defined until the end of the function.
- Calling `myfunct()` on line 6 gives: results in printing 300

Local scope

```python
def myfunc():
    # local scope
    x = 300
    print(x)

myfunc()
# print function 2
print("The result is: ", x)
```

- First prints 300

Local scope

```
def myfunc():
    # local scope
    x = 300
    print(x)

myfunc()
# print function 2
print("The result is: ", x)
```

- First prints 300
- Then we get an error `NameError: name 'x' is not defined` because 'x' is only defined in the local (inner) scope

Enclosing scope

```
def myfunc():
  # outer scope
  x = 300
  def myinnerfunc():
    # inner scope
    print(x)
    y = 100
  myinnerfunc()
  print(y)

myfunc()
```

- Enclosing scope: Outer's variables have a larger scope and can be accessed from the enclosed function `myinnerfunc()`.
- However, y is not defined outside of the inner/local scope. Therefore, the function prints 300, but then raises an Error `NameError: name 'y' is not defined`

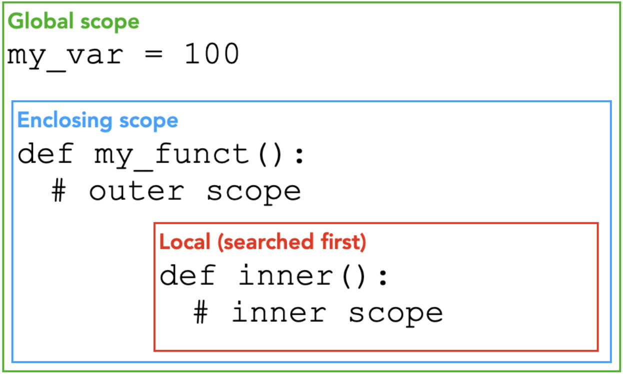
Global scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.
Global variables are available from within any scope, global and local.

```
x = 300

def myfunc():
  print(x)

myfunc()

print(x)
```

- Prints 300 and
- 300

Summary: Python scope



Naming variables -

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

- prints ??

Global keyword -

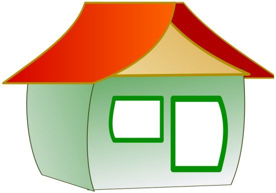
```
x = 300

def myfunc():
  global x
  x = 200
  print(x)

myfunc()

print(x)
```

- ???

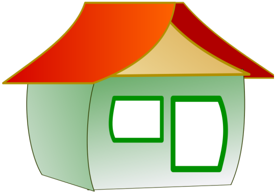
Object-oriented Programming

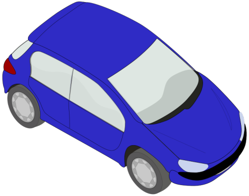
Objects

What is an object?

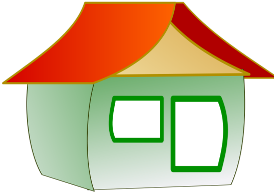


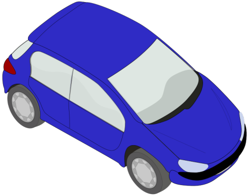
What is an object?



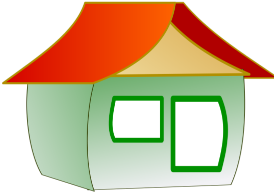


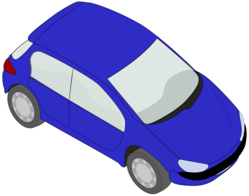
What is an object?





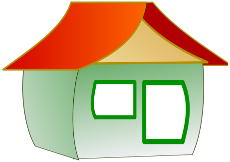
What is an object?





Object Properties I

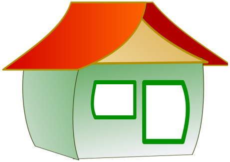
Objects have properties/characteristics or a **state**:



- color: green
- number of doors: 1
- number of windows: 1
- flat roof: no

Object Properties II

Objects have properties/characteristics or a **state**:

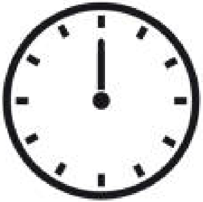


- color: green
- number of doors: 1
- number of windows: 1
- flat roof: no

- changeable properties, e.g., color, current speed, time, battery stand, etc.
- (almost) unchangeable properties, e.g., number of tires, model number, number of needles, monitor size, etc.

Messages to Objects I

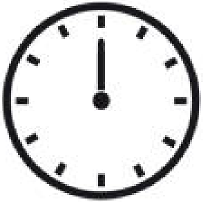
Set the time for 6pm!



$\Rightarrow$

Messages to Objects I

Set the time for 6pm!



$\Rightarrow$

Messages to Objects II

Switch on!

$\Rightarrow$

Switch on!

$\Rightarrow$

- different actions from different messages:
  - simple setting of a property, e.g., time, color, etc.
  - checking of a value and changing a property accordingly, e.g., speed, max. speed, etc.
- messages through **method** calling

Messages to Objects

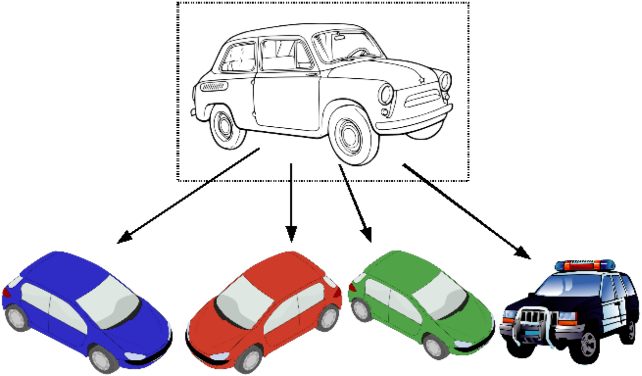
- methods/messages can be parameterized, e.g.,
  - new color
  - new time
  - new speed
- or unparameterized, e.g.,
  - Switch on!
  - Light on!
- method can output results, the so-called return values, e.g.,
  - new speed after accelerating
  - did the switching-on work
  - battery stand

Terms

- **Classes** describe concepts (objects)
- **methods** are parts of classes
- **instance(s)** a concrete realization of an object (polymorphism)
- Example: a class Dog (describes properties of the concept dog); instances are concrete instantiations of objects: dogA, dogB,...

Classes

Classes = Construction Plans (Polymorphism)

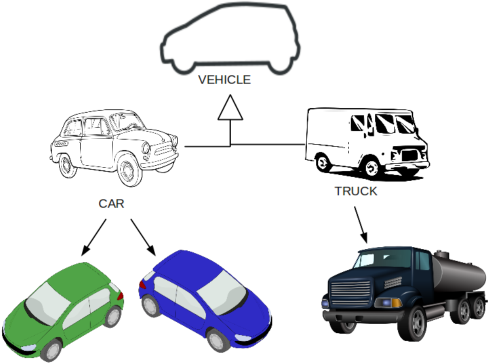


- **Classes**: construction plans for objects or types of objects, e.g., there is the class *car* containing different objects with different properties (polymorphism allows an concept to take on multiple forms)
- objects of a class have the same basic structure (e.g., all cars have four tires), but can also differ in some aspects, e.g., the color, the size, etc.

Classes = Construction Plans

- we can create instances of a class $\Rightarrow$ objects = instances of a class
- classes define *attributes* (properties) or *instance variables*:
  - color - has to be set
  - speed - has to be set
  - battery stand - has a default value
- objects fill the instance variables with specific values $\Rightarrow$ state
  - time = 10:10 or time = 6:15
  - color = green or color = blue or color = red
  - floors = 1 or floors = 10
- classes define methods, which can change the state of the objects
- classes = templates

Inheritance

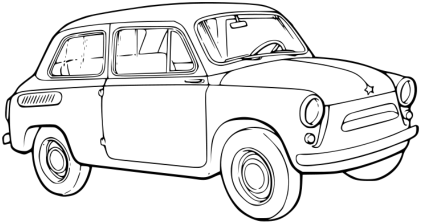


the object *blue car* belongs to the class *car*, which belongs to the class *vehicle*: all blue cars are cars and all cars are vehicles $\neq$ all vehicles are cars – there are also trucks, bulldozers, etc.

- different objects might partially have the same behavior/characteristics
- inheritance $\rightarrow$ avoids code redundancy

Object Composition

objects can be composed of different objects:



Details and Application in Python

Software-Objects = Real-life Objects

Attributes/Instances	`annesAccount`	`stefansAccount`
id	1	2
holder	'Anne'	'Stefan'
balance	200	1000

Attributes

- describe the *state* of the object
- contain the *data* of an object
- can change with time

Classes = Construction Plans

```
# content of script myacc.py
class Account:
    """ a class for objects """

if __name__ == "__main__":
    annesAccount = Account()
    stefansAccount = Account()
    print(type(annesAccount))
    print(type(stefansAccount))

>>> python myacc.py
<class '__main__.Account'>
<class '__main__.Account'>
```

Object Types: type function

Remember, all values in Python have types that you can retrieve with the type function:

- str, float etc (basic types)
- type also works for classes

```
>>> type("hello")
<class 'str'>
>>> type(10)
<class 'int'>
>>> type(11.11)
<class 'float'>
>>> type(annesAcc)
<class '__main__.Account'>
```

UML Class-Diagrams

- Unified Modeling Language
- Visualization standard for object-oriented programming (and more)
- Helps us to visualize the functionality we want to implement

Name of the class
Attributes
Methods

Account
id holder balance
print_info() deposit(amount) withdraw(amount)

Attributes must exist, when we want to access them

```python
class Account:
    def print_info(self):
        print("Balance:", self.balance)

if __name__ == "__main__":
    stefansAcc = Account()
    stefansAcc.print_info()
```

Why do we get an error with this code?

Initialization / Constructor

```
class Account:
    # CONSTRUCTOR
    def __init__(self):
        self.balance = 0

    # METHODS
    ...

if __name__ == "__main__":
    annesAcc = Account()
    stefansAcc = Account()
```

- **Constructor** `__init__(self)` is always automatically called when an object of the class is created
- Used to assign to attributes of an object some initial/default values

Initialization / Constructor

```python
class Account:
    # CONSTRUCTOR
    def __init__(self, num, person):
        self.balance = 0
        self.id = num
        self.holder = person
    # METHODS
    ...
if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    stefansAcc = Account(2, "Stefan")
```

- Example **constructor** with required positional arguments (id, name)
- What happens if we would call **a = Account()**?

Initialization / Constructor

```python
class Account:
    # CONSTRUCTOR
    def __init__(self, num, person):
        self.balance = 0
        self.id = num
        self.holder = person
    # METHODS
    ...
if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    stefansAcc = Account(2, "Stefan")
```

- Example **constructor** with required positional arguments (id, name)
- What happens if we would call **a = Account()**?
- **a = Account()** now raises a TypeError: missing 2 requires positional arguments

The `self` keyword in Python OOP

Why do we have `self`?

Initialization / Constructor

```
annesAcc = Account(1, "Anne")
```

1. the constructor of `Account` is called;
   the variable `self` points now to the new **object itself**;
2. through the constructor the object gets initialized
   (the attributes get assigned the given or the default values)
3. assignment of the newly created object to the variable `annesAcc`

Classes = Construction Plans

```python
class Account:
''' a class for objects
representing an account '''
  [...] # assume constructor
# Main part of the program
if __name__ == "__main__":
  # Creating objects
  annesAcc = Account(1,"Anne")
  stefansAcc = Account(2,"Stefan")
  # Accessing attributes
  print(annesAcc.balance)
 # Assigning attributes
  annesAcc.holder = "Anne Li"
  # good way to assign attributes?
```

- class names start with capital letters
- objects are instantiated/created by calling the class
- assignment of/access on attributes and methods through the *dot notation*

Methods = Functions that belong to a class

```python
class Account:
    [...] # assume constructor
    # METHODS
    def deposit(self, amount):
        self.balance += amount

if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    annesAcc.deposit(500)
```

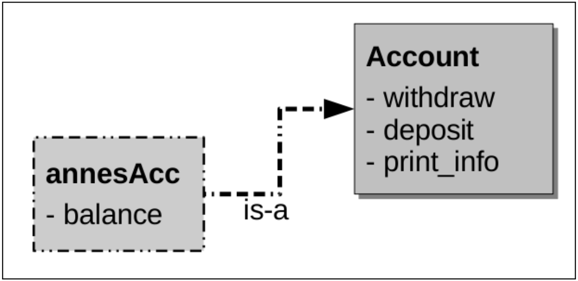
Instance Methods

- operate on objects that were created from a class (constructor page 44)
- our code changes the attributes of an object or allows access to them
- first parameter: `self` (convention)

Methods = Functions that belong to a class

```python
class Account:
    [...] # assume constructor
    # METHODS
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
    def print_info(self):
        print("Balance:", self.balance)

if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    annesAcc.deposit(500) # better way
    annesAcc.withdraw(20)
    annesAcc.print_info()
```

Objects are linked with their class



- `annesAcc.deposit(500)`
- first step: search for the method in the object itself (technically, methods for individual objects can also be defined within the object itself – practically, methods are defined in the class, see second step)
- second step: search the method in the class from which the object originates

Class Design

Rules for good class design

1. How can I describe the state of my object? ⇒ *attributes*
2. What do I know about my object before or during its creation? ⇒ *constructor*
3. What operations that change the state of the object might need to be applied later on? ⇒ *instance methods*

Only manipulate attributes through instance methods

Bad coding (directly accessing instance attributes):

```
stefansAcc.balance = 1000
```

Data encapsulation:

- attributes of an object should be “hidden"from “external"manipulations (= from code that is used by the object itself)
- attributes of an object should only be manipulated from code that was defined within the class
- this makes sure that the state of an object is always valid

Example

- account balance cannot be negative
- Stefan’s account balance is €1000, he wants to withdraw €1500
- bank clerk gives him the money and Stefan’s balance is at -€500 ⇒ bank manager is angry!

```python
class Account:
    ...
    # METHODS
    def withdraw(self, amount):
        if amount > self.balance:
            amount = self.balance
        self.balance -= amount
        return amount
    ...

if __name__ == "__main__":
    stefansAcc = Account(2, "Stefan")
    stefansAcc.deposit(1000)
    cash = stefansAcc.withdraw(1500)
    print("Oh no, I only got:", cash)
```

Setter Methods: Change the Attribute Values

```python
class Account:
    def set_holder(self, person):
        self.holder = person

if __name__ == "__main__":
    stefansAcc = Account(2, "Stefan")
    stefansAcc.deposit(1000)
    stefansAcc.set_holder("Andrea")
```

- for each attribute that needs to be changed from outside the class, we have to create a **setter method**
- allows validation

Example of Validation in a setter method:

```
def set_holder(self, person):
    if (not type(person) == str):
        raise TypeError
    if not person.strip().split() > 1):
        print("Give a valid non-empty name")
        raise ValueError
    self.holder = person
```

Coding Style

1. assign values to the attributes only in instance methods (setter methods) or in the constructor
2. change the values of the attributes only through setter methods
3. access (read-only) on the values of the attributes through `print(stefansAcc.balance)` is OK (but we can also define getter methods)

String Representation of an Object

```
class Account:
  def __repr__(self):  %  __str__ alternative
    res = "*** Account Info ***\n"
    res += "Account ID:" + str(self.number) + "\n"
    res += "Holder:" + self.holder + "\n"
    res += "Balance: " + str(self.balance) + "\n"
    return res   % N.B. string formatting is preferred

if __name__ == "__main__":
  annesAcc = Account(1, "Anne")
  annesAcc.deposit(200)
  print(annesAcc)
```

- **Magic/Special methods** = (magic) methods that are called from Python in specific cases
- here: we need a string representation of the object, e.g., `print(annesAcc)` or `str(annesAcc)`

Inheritance in Python

```python
class Animal:
    def make_sound(self):
        print('ROAR')
# Uses the make_sound method of its parent.
class Lion(Animal):
    pass
class Duck(Animal):
        # Overrides the make_sound method of its parent.
    def make_sound(self):
        print('QUACK')

if __name__ == "__main__":
    lion = Lion()
    lion.make_sound() # prints ROAR
    duck = Duck()
    duck.make_sound() # prints QUACK
```

Summary

- classes represent concepts (data with operations)
- instances are concrete realizations of the classes
- instances are created through the class constructor
- methods allow for the encapsulation of the attributes
- attributes of instances should only be changed in setter methods or in the constructor
- setter methods (and constructors) can be used for validation
- inheritance should be used to avoid redundant code

Python fundamentals

- Lists, List indexing and slicing, multidimensional lists
- List comprehension
- Copying lists: Shallow vd Deep* copy
- String formatting*
- Scripts & Command-Line Arguments *

* = covered in lab

Lists: Indices

```
myList = ["a", "b", "c", "d", "hello"]
```

0	1	2	3	4
"a"	"b"	"c"	"d"	"hello"
-5	-4	-3	-2	-1

What is the output of the following code?

```
print(myList[1])
print(myList[4])
print(myList[-2])
```

String Immutability

- Strings are sequences of characters
- We can access individual characters of a string

```
>>> myString = "telephone"
>>> print(myString[2])
l
>>> print(myString[4:]) # copy!
phone
```

- Strings are immutable: not changeable sequences
  `myString[0] = "T"` ⇒ **DOES NOT WORK!**
- N.B. String concatenation creates a new string object, i.e.
  `myString = "T"+ myString[1:]`

Lists: Indices

```
myList = ["a", "b", "c", "d", "hello"]
```

0	1	2	3	4
“a"	"b"	"c"	"d"	"hello"
-5	-4	-3	-2	-1

What is the output of the following code?

```
print(myList[4][1])
```

Lists: Slicing

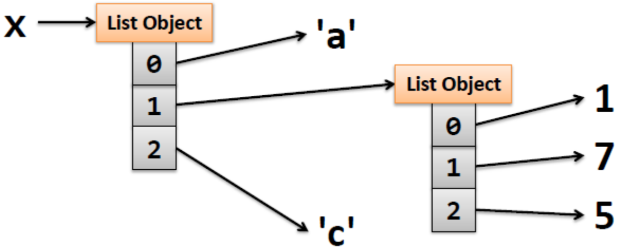
'a'	'b'	'c'	'd'
0	1	2	3

creates **copies** of list objects

```
>>> myList = ['a', 'b', 'c', 'd']
>>> myList[0:3]
['a', 'b', 'c']
>>> myList[2:3]
['c']
>>> myList = ['a', 'b', 'c', 'd']
>>> myList[2:4]
['c', 'd']
>>> myList[1:]
['b', 'c', 'd']
>>> myList[:3]
['a', 'b', 'c']
>>> myList[:]
['a', 'b', 'c', 'd']
```

Multidimensional lists

```
x = ['a', [1, 7, 5], 'c']
print("x[0] is: {}".format(x[0]))
print("x[1] is: {}".format(x[1]))
print("x[2] is: {}".format(x[2]))
print("x[1][0] is: {}".format(x[1][0]))
print("x[1][1] is: {}".format(x[1][1]))
print("x[1][2] is: {}".format(x[1][2]))
```



Multidimensional lists

```
myList = ["a", "b", [1, 2, 3], "d", "e"]
myList[3] = [4, 5, 6]
```

0	1	2	3	4
"a"	"b"	$[1, 2, 3]$	$[4, 5, 6]$	"e"

a) What output do we get?

```
print(myList[1])
print(myList[2][0])
```

b) How can we access '5' ?

c) What output do we get?

```
print(myList[5])
print(myList[2][3])
```

Shared References

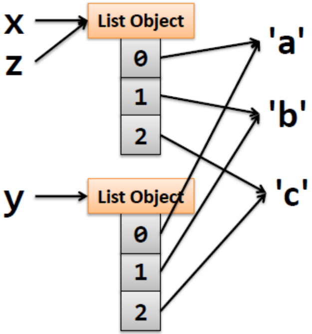
- Variables do not directly contain the values (like a tupperware contains food)
- Variables point to the position in memory and the position in memory keeps the values (like a name points to a person, but the name does not contain the person)

```
>>> michael = ["engineer", "germany", "40"]
>>> mike = michael
>>> thomson = michael
>>> michael
["engineer", "germany", "40"]
>>> mike
["engineer", "germany", "40"]
>>> thomson
["engineer", "germany", "40"]
```

Lists: Copy (Shallow Copy)

- slicing creates a shallow copy of a list
- a shallow copy creates a new list object and adds references to the same object, the original lists had (http://docs.python.org/3.2/library/copy.html)
- the *is*-operator can be used to check if two variables point to the same object

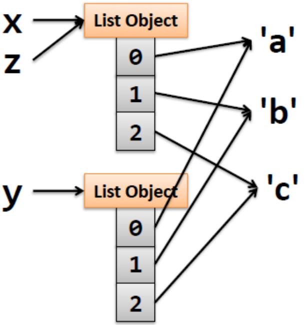
```
>>> x = ['a', 'b', 'c']
>>> z = x
>>> y = x[:]
>>> y
['a', 'b', 'c']
>>> z is x
True
>>> y is x
False
```



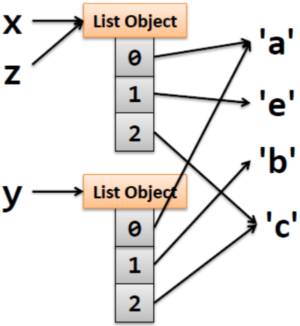
Lists: Copy (Shallow Copy)

```
>>> x = ['a', 'b', 'c']
>>> z = x
>>> y = x[:]
>>> y
['a', 'b', 'c']
```

```
>>> x[1] = 'e'
>>> x
['a', 'e', 'c']
>>> y
['a', 'b', 'c']
>>> z
# what is printed?
```



$\Rightarrow$



Comprehension

- Create a new list from existing list
- Computationally more efficient than equivalent for loop statement
- Also: create a new dict from an existing dict with comprehension

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> subset = [i for i in numbers if i > 5]
>>> subset
[6, 7, 8, 9]
```

Summary

- Lists
- Mutable vs. immutable types, String formatting
- Slicing
- Multidimensional lists
- Lists: copying
- More Python details in exercise hour:
  - Python copying lists (shallow and deep copy)
  - Values and references
  - Scripts
  - Command-line arguments and ArgParse

Questions?

Next: Object-Oriented Programming I

Next time: Object-Oriented Programming II

Questions

Questions?

Next Time: Object-oriented Programming II

Credits

Slides adapted from prior version by Katerina Kalouli, Annemarie Friedrich, Benjamin Roth, Florian Fink.