Python: objects, classes

Intro. to computing

`https://mandarm.github.io/computing.html`

Objects

- **Object:** any data item, i.e., value
  e.g., numbers (integers, floats), sequences, strings, dictionaries, sets, etc.
- **References:** names used in a Python program for the above

Properties of objects: each object has a type $\Rightarrow$

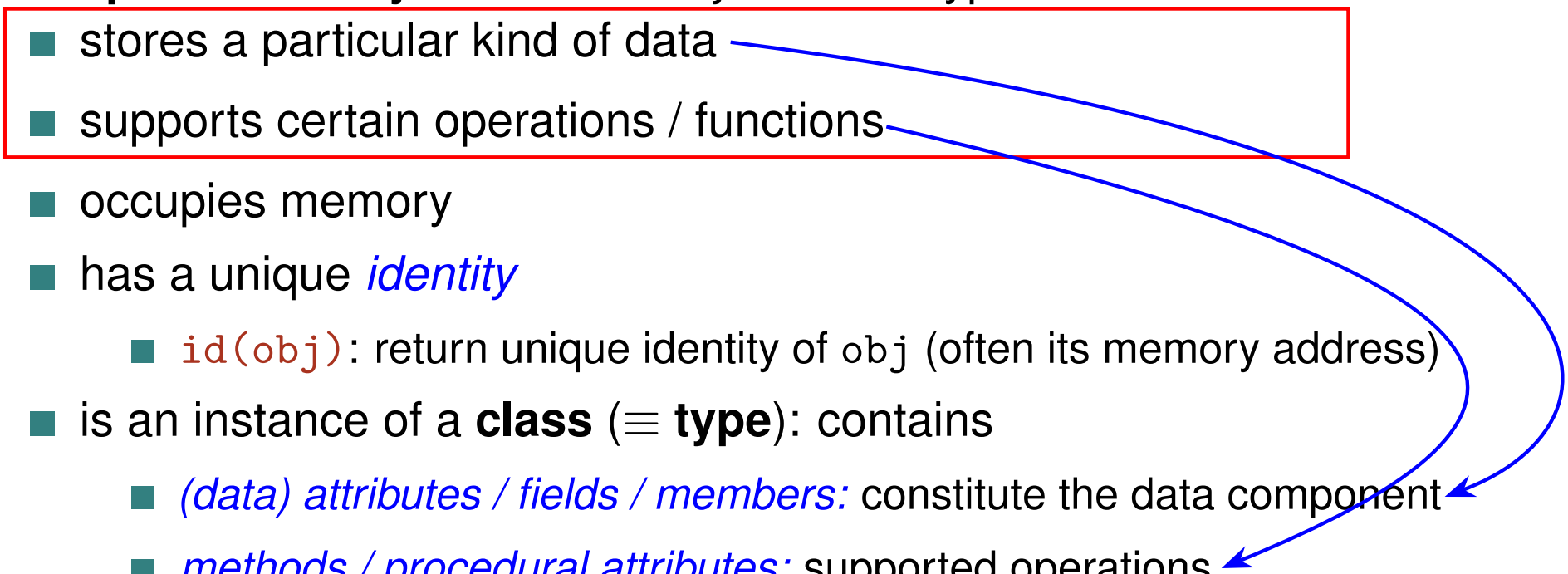
- stores a particular kind of data
- supports certain operations / functions
- occupies memory
- has a unique *identity*
  - `id(obj)`: return unique identity of `obj` (often its memory address)

Objects

- **Object:** any data item, i.e., value
  e.g., numbers (integers, floats), sequences, strings, dictionaries, sets, etc.
- **References:** names used in a Python program for the above

Properties of objects: each object has a type $\Rightarrow$

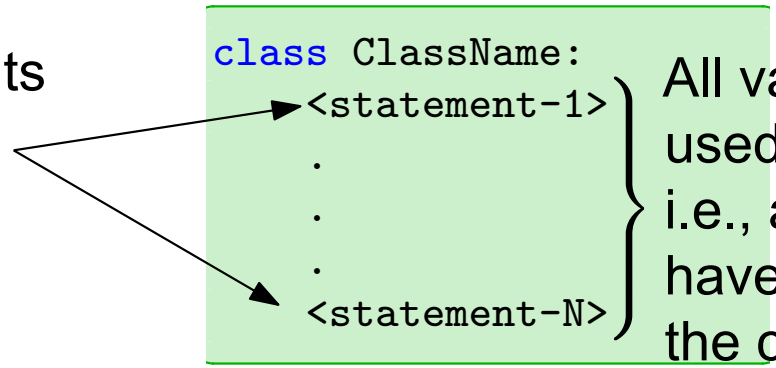
- stores a particular kind of data
- supports certain operations / functions
- occupies memory
- has a unique *identity*
  - `id(obj)`: return unique identity of `obj` (often its memory address)
- is an instance of a **class** ($\equiv$ **type**): contains
  - *(data) attributes / fields / members:* constitute the data component
  - *methods / procedural attributes:* supported operations



Our convention: attributes $\equiv$ data, methods $\equiv$ functions

Creating a class I

These statements usually define attributes and functions.




```
class Point1:
    x = 0
    y = 0
```

```
class Point2:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'X coordinate: {self.x};    Y coordinate: {self.y}.'
```

Creating a class II

Example:

```
p, q = Point1(), Point1()   # Create objects / instances of the class
p.x, q.y = 10, 100          # Change attribute values
p2 = Point2(2,6)            # More flexible object creation
p3 = Point2(y=9,x=3)        # More flexible object creation
print(p, p.x, p.y)          # print(object) is not helpful
print(q.x, q.y)
print(f'{p2}\n{p3}')        # without appropriate __repr__() method
```

Observations:

- `ClassName()` creates an object.
- Can write `__init__()` function for more customised object creation.
- Can write `__repr__()` to print object in a desired 'pretty' format (must *return* a string, **NOT** print it).

Calling methods (functions)

- Using class name: called as usual, no special handling
- Using an object: object is automatically inserted as the first argument; all others shifted right by one position.

Exercise: Which of the following statements are accepted by the Python interpreter without errors? Why?

```
class Dummy:
    def f():
        print("hi")
    def g(*everything):
        print(*everything)
```

```
Dummy.f()
Dummy.g(1,2,3)
a = Dummy()
a.f()           # TypeError
a.g(1,2,3)
```

no. of args mismatch

`Dummy.f` is a **function**, `a.f` is a **method**.

Defining methods

- Depends on intended use
- If attributes are not referenced, define as usual.
- If attributes are referenced, use first argument to represent object itself
  - *conventionally* called `self` (but can be called anything else)
  - reference attributes as `self.attribute_name` (no shortcuts)

Exercise: Which of the following statements are accepted by the Python interpreter without errors? Why?

```
class Dummy:
    x = 0.5
    def mysin():
        import math
        print(math.sin(math.pi * x))
    def mycos(self):
        print(math.cos(math.pi * self.x))
```

```
a = Dummy()
a.mysin()       # TypeError
a.mycos()
Dummy.mysin()   # NameError
Dummy.mycos(a)
```

no. of args mismatch

x not defined;
`global x` does not help

Attributes I

```
class Dummy:
    x = 0.5
    l = []
    def __init__(self, x=1, lst=[]):
        self.x = x
        self.lst = lst
    def __repr__(self):
        return f"x={self.x},  l={self.l},  lst={self.lst}\n"
```

```
a = Dummy(lst=["a"])
a.l.append("z")
a.lst.append("y")
b = Dummy(2, ["b"])
c = Dummy(3, ["c"])
print(f'{a}{b}{c}')
print(f'{Dummy.x}')
```

```
del b.lst        # Attributes can be deleted!
print(b)         # AttributeError
b.lst = ["hi"]   # Attributes can be added
print(b)
b.xyz = "???"    # Attributes can be added
print(b, b.xyz) # xyz added to b only, not a, c
del Dummy.l      # Try printing a, b, c now
```

Attributes II

```
Dummy.l = []          # Adding l back
Dummy.new = 123456 # Adding a new attribute
print(f'{a}{b}{c}{a.new} {b.new}')
a.new = 700           # Reassignment, i.e., only a changes
a.l.append("all")   # Mutating => a.l, b.l, c.l changes
a.l = 98765           # Reassignment
```

Remember:

immutable vs. mutable
mutating vs. assignment

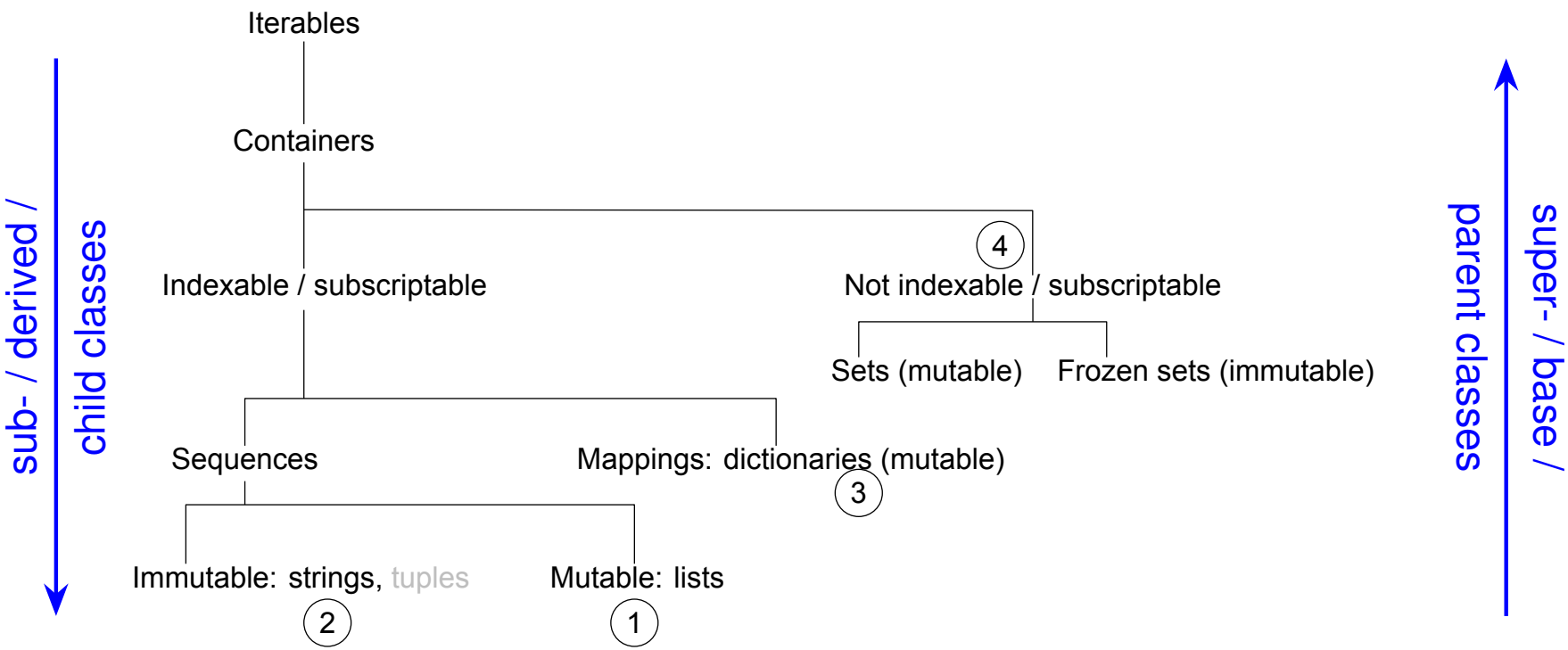
Observations

- Attributes can be added and removed dynamically to objects, or a class.
- Digression: use `dir(classname)` or `dir(objectname)` to see all attributes of class / object.
- **Class variables**: common to all instances (objects), e.g., `Dummy.l`
- **Instance variables:** local to individual objects, e.g., `a.lst`

  $\Rightarrow$ `Dummy.lst` does not exist.
- Instance variables *mask* class variables with the same name.

Inheritance

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```



How inheritance works

- Attribute / method lookup proceeds up the class hierarchy until match is found
- Subclass may add / *override* (replace) attributes, methods of baseclass / superclass
- Subclass may *extend* method of baseclass / superclass

```
class DerivedClassName(BaseClassName):
    def methodname(self, arguments):
        BaseClassName.methodname(self, arguments)
        <additional processing>
    ...
```

  - holds true for `__init__` as well, i.e., `__init__` of derived class does not automatically invoke `__init__` of base class

Multiple inheritance: will not cover

Private members: the problem

```
1  class Mapping:
2      def __init__(self, iterable):
3          self.items_list = []
4          self.update(iterable)
5
6      def update(self, iterable):
7          for item in iterable:
8              self.items_list.append(item)
9
10 class MappingSubclass(Mapping):
11     # __init__ inherited from baseclass
12     # update method overriden
13     def update(self, keys, values):
14         for item in zip(keys, values):
15             self.items_list.append(item)
```

- Assumption: call to `update` in line 4 refers to `Mapping.update`
- What happens during the following?

```
1  x = Mapping([1,2])
2  y = MappingSubclass([1,2])
3  y = MappingSubclass([1,2],
       [3,4])
```

Private members: the problem

```
1  class Mapping:
2      def __init__(self, iterable):
3          self.items_list = []
4          self.update(iterable)
5
6      def update(self, iterable):
7          for item in iterable:
8              self.items_list.append(item)
9
10 class MappingSubclass(Mapping):
11     # __init__ inherited from baseclass
12     # update method overriden
13     def update(self, keys, values):
14         for item in zip(keys, values):
15             self.items_list.append(item)
```

- Assumption: call to `update` in line 4 refers to `Mapping.update`
- What happens during the following?

```
1  x = Mapping([1,2])
2  y = MappingSubclass([1,2])
3  y = MappingSubclass([1,2],
       [3,4])
```

- Line 2: `TypeError` for `Mapping.__init__()`
- Line 3: `TypeError` for `MappingSubclass.update()`

Adapted from `https://docs.python.org/3/tutorial/classes.html#private-variables`

Private members: the solution I

Name mangling: *any* occurrence of an identifier of the form `__identifier` (≥ 2 leading underscores, ≤ 1 trailing underscore) is *textually replaced* with `_classname__spam`, where `classname` is the current class name (with leading underscore(s) stripped).

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of update() method
```

Private members: the solution II

```
1  class MappingSubclass(Mapping):
2      # __init__ inherited from baseclass
3      # update method overriden
4      def update(self, keys, values):
5          for item in zip(keys, values):
6              self.items_list.append(item)
7
8  x = Mapping([1,2])
9  y = MappingSubclass([1,2])         # Works, but probably not intended
10 y = MappingSubclass([(1,2)])       # Works as intended
11 y = MappingSubclass([1,2], [3,4]) # TypeError
12 y.update([1,2], [3,4])            # Works as intended
```

Utility functions (built-in)

- `isinstance(obj, classname)`: returns `True` if and only if `obj` in an instance of `classname` or a class derived from it
  - NOTE: difference with `type()`

```
x = True
isinstance(x, int)  # => True
type(x)             # => bool
type(x) == int      # => False
```

- `issubclass(c1, c2)`: returns `True` iff `c1` is a subclass of `c2`
  e.g., `issubclass(bool, int)` returns `True`