COL100: Introduction to Computer Science

8: Data and objects

Beyond built-in types

Built-in types: `int`, `bool`, `float`, strings, tuples, lists, dicts, …

In practice, information usually has internal *structure*, meaningful *relationships* between different pieces

We can represent this structure by defining our own types, called **classes**.

Example: Your time table

COL100
Introduction to Computer Science
Lectures: Mon 9:30-11:00, Thu 9:30-11:00
Tutorials: none
Labs: Wed 13:00-15:00

MTL100
Calculus
Lectures: Tue 9:00-10:00, Wed 9:00-10:00, Fri 9:00-10:00
Tutorials: Mon 14:00-15:00
Labs: none

…

```
tt = [
  ( "COL100",
    "Introduction to Computer Science",
    [("Mon", 9.5, 11), ("Thu", 9.5, 11)],
    [],
    [("Fri", 13, 15)] ),
  ( "MTL100",
    "Calculus",
    [("Tue", 9, 10), ("Wed", 9, 10), ("Fri", 9, 10)],
    [("Mon", 14, 15)],
    [] ),
  …
]

tt[0]           # ("COL100", …)
tt[0][2][-1][0] # "Thu"
```

Example: Your time table

COL100
Introduction to Computer Science
Lectures: Mon 9:30-11:00, Thu 9:30-11:00
Tutorials: none
Labs: Wed 13:00-15:00

MTL100
Calculus
Lectures: Tue 9:00-10:00, Wed 9:00-10:00, Fri 9:00-10:00
Tutorials: Mon 14:00-15:00
Labs: none

…

Example: Your time table

COL100
Introduction to Computer Science
Lectures: Mon 9:30-11:00, Thu 9:30-11:00
Tutorials: none
Labs: Wed 13:00-15:00

MTL100
Calculus
Lectures: Tue 9:00-10:00, Wed 9:00-10:00, Fri 9:00-10:00
Tutorials: Mon 14:00-15:00
Labs: none

…

Course

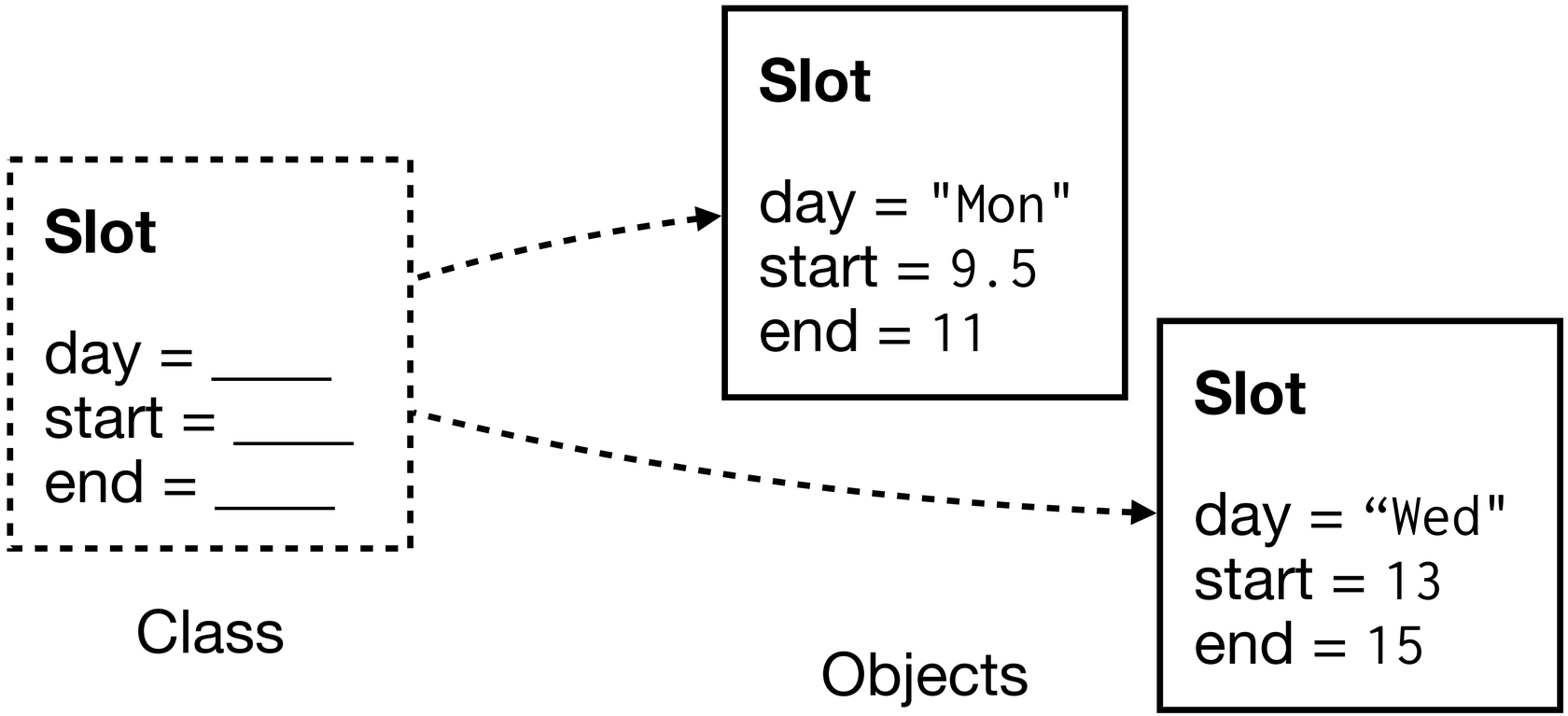
code = _____
title = _____
lectures = [_____]
tutorials = [_____]
labs = [_____]

Slot

day = _____
start = _____
end = _____

Classes and objects

A **class** is a programmer-defined type which specifies certain **attributes**.
Using a class we can create **objects**, which are **instances** of the class.

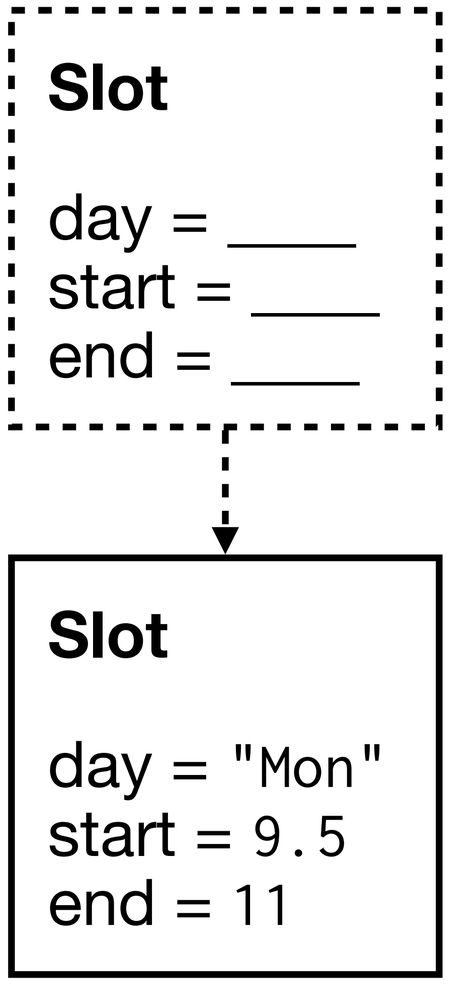


Classes and objects in Python

```
class Slot:
  def __init__(self, day, start, end): # constructor
    self.day = day
    self.start = start
    self.end = end

slot1 = Slot("Mon", 9.5, 11)

slot1.day   # "Mon"
slot1.start # 9.5
slot1.end   # 11
```



```
col100 = Course(
  "COL100",
  "Introduction to Computer Science",
  [Slot("Mon", 9.5, 11), Slot("Thu", 9.5, 11)],
  [],
  [Slot("Fri", 13, 15)])
mtl100 = Course(
  "MTL100",
  "Calculus",
  [("Tue", 9, 10), ("Wed", 9, 10), ("Fri", 9, 10)],
  [("Mon", 14, 15)],
  [])
…
tt = [col100, mtl100, …]

tt[0]                    # <…Course object at 0x…>
tt[0].lectures[-1].day # "Thu"
```

Methods

With only data attributes, objects are not so different from dictionaries…

```
slot1 = {"day": "Mon", "start": 9.5, "end": 11}
```

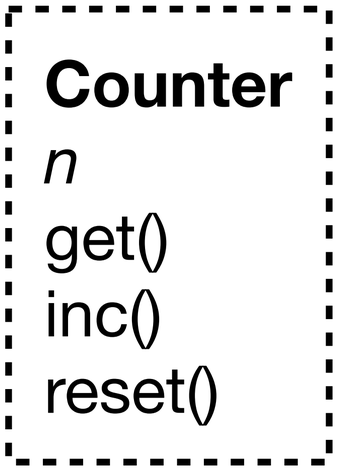
But objects can also contain attributes that are functions! These are called **methods**, and can act on the data attributes of the object.

```
class Slot:
  def __init__(self, day, start, end):
    self.day = day
    self.start = start
    self.end = end
  def duration(self):
    return self.end - self.start
```

```
slot1 = Slot("Mon", 9.5, 11)

slot1.duration() # 1.5
```

Example: Counters




```
class Counter:
  def __init__(self):
    self.n = 0
  def get(self):
    return self.n
  def inc(self):
    self.n += 1
  def reset(self):
    self.n = 0
```

```
cmps = Counter()
swaps = Counter()

def mySortingFunction(a):
  for i in range(len(a)):
    …
    cmps.inc()
    if (a[i] > a[j]):
      a[i], a[j] = a[j], a[i]
      swaps.inc()
      …

mySortingFunction(aList)
print(cmps.get(), swaps.get())
```

Example: Rational numbers

```
class Rational:
  def __init__(self, p, q):
    self.p = p
    self.q = q
  def add(self, r):
    p = self.p * r.q + self.q * r.p
    q = self.q * r.q
    g = gcd(p, q)
    return Rational(p // g, q // g)
  …
```

Rational
p, *q*
add(*r*)
sub(*r*)
mul(*r*)
div(*r*)
toFloat()

Now the essential functionality needed to work with a rational number is included within the object itself!

```
fiveSixths = Rational(1,2).add(Rational(1,3))
```

Example: Rational numbers

```
class Rational:
  def __init__(self, p, q):
    g = gcd(p, q)
    self.p = p // g
    self.q = q // g
  def add(self, r):
    return Rational(self.p * r.q + self.q * r.p, self.q * r.q)
  …
```

Invariants (e.g. rational is in simplest form) should be enforced by the constructor itself. Then you can't create an object that violates them

Example: Rational numbers

You can use `isinstance(object, class)` to check if an object is the right type:

```
class Rational:
  def __init__(self, p, q):
    g = gcd(p, q)
    self.p = p // g
    self.q = q // g
  def add(self, r):
    if isinstance(r, Rational):
      return Rational(self.p * r.q + self.q * r.p, self.q * r.q)
    elif isinstance(r, int):
      return Rational(self.p + self.q * r, self.q)
    else:
      raise TypeError
  …
```

Exercises

- Complete the implementation of `Rational`. Make sure it works for negative inputs too, e.g. `Rational(1,-2)` should have $p = -1$, $q = 2$.
- Implement a complex number type `Complex` with similar functionality. Instead of `toFloat()`, implement methods `abs()` and `arg()`.

Note: To get some useful mathematical functions into your scope, type e.g.

```
from math import gcd, sqrt, atan2
```

at the top of your program. See https://docs.python.org/3/library/math.html

Python-specific: Special methods

In Python, some method names are special. e.g. `__init__` is always the constructor

- `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`: called for <, <=, ==, !=, >, >= e.g. `obj1 < obj2` is evaluated by calling `obj1.__lt__(obj2)`
- `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`: called for +, -, *, /, //
- `__repr__`, `__str__`: called when displaying an object
  - `__repr__` should be as informative and unambiguous as possible
  - `__str__` can be more readable and user-friendly

```
class Rational:
  …
  def __add__(self, r):
    if isinstance(r, Rational):
      return Rational(self.p * r.q + self.q * r.p, self.q * r.q)
    …
  def __repr__(self):
    return 'Rational({},{})'.format(self.p, self.q)
  def __str__(self):
    return '{}/{}'.format(self.p, self.q)
  …

r = Rational(1,2) + Rational(1,3)
r         # Rational(5,6)
print(r) # 5/6
```

Exercises

- Implement all the other special methods for `Rational` (except `__floordiv__`).