# 1. The Python Data Model

### **How Special Methods Are Used**

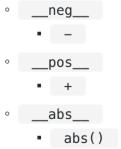
- · meant to be called by Python interpreter and not programmer
- · can be used to emulate numeric types with operator overloading
- can be used to set how a custom type is printed ( repr )
  - str is called by the str() constructor and implicitly used by the print function
  - str should return a string suitable for display to end users
  - · choose repr if only implementing one because when no custom str is available Python will call repr as a fallback
- can be used for arithmetic operators ( add , mul )
- can be used to use custom type as a bool value ( bool )

#### C

Overview of Special Methods		
string/bytes representation		
<pre>    repr </pre>		
°str		
°format		
obytes		
conversion to number		
。abs		
obool		
°complex		
°int		
·float		
<pre>hash</pre>		
<pre>oindex</pre>		
collection-like functionality		
。len		
∘getitem		
°setitem		
°delitem		
·contains		
iteration		
<pre>oiter</pre>		
o reversed		

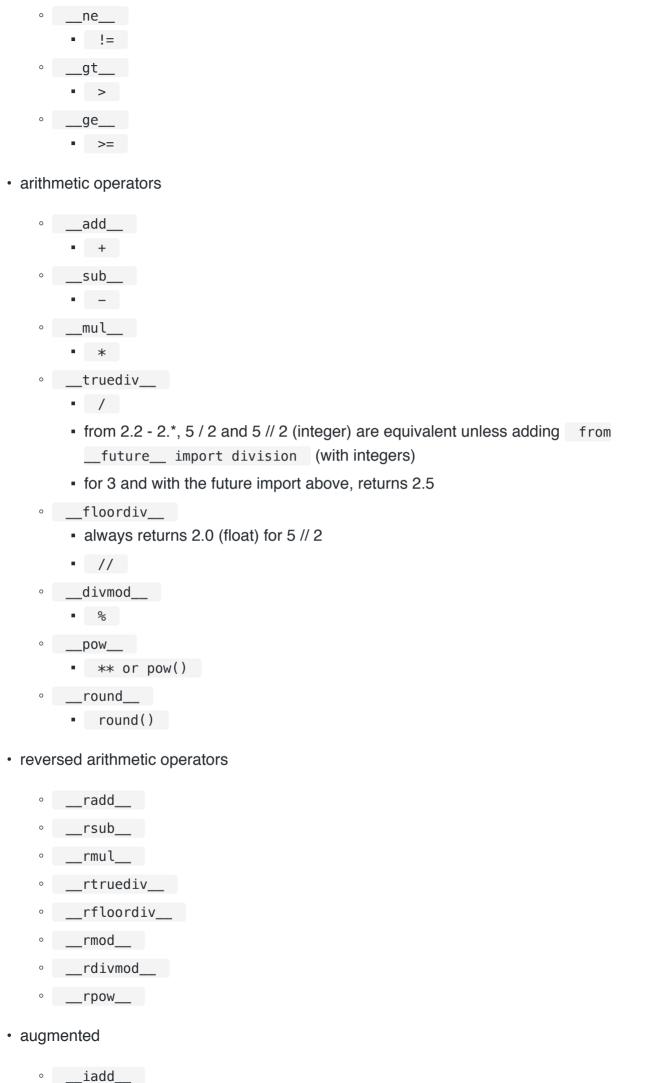
C	next
• ma	king an object callable
C	call
• cor	ntext management
C	enter
C	exit
• ins	tance creation and destruction
(	new
C	init
(	odel
• attı	ribute management
C	getattr
C	<pre>getattribute</pre>
(	setattr
(	odelattr
(	dir
• attı	ribute descriptors
C	get
C	set
(	delete
• cla	ss services
(	prepare
(	instancecheck
(	subclasscheck
Opera	ators
• una	ary numeric operators
C	neg
	noc

## 



• rich comparison operators

0		lt
	•	<
0		le
	•	<=
0		eq
	•	==



o \_\_isub\_\_\_

0	imul
0	itruediv
0	ifloordiv
0	imod
0	ipow
• bitwi	se operators
• bitwi	se operatorsinvert
٥	invert
٥	invert

。 \_\_\_and\_\_\_

or\_\_\_or\_\_

• &

° \_\_xor\_\_

· reversed bitwise operators

\_\_rlshift\_\_\_\_rrshift\_\_\_\_rand\_\_\_\_ror\_\_\_\_rxor

· augmented bitwise operators

\_\_ilshift\_\_\_\_irshift\_\_\_\_iand\_\_\_\_ior\_\_\_\_ixor\_\_

- len is not called as a method because it gets special treatment as part of the Python data model, just like abs
- can also make len work with custom objects using the special method len
- special methods allow custom objects to behave like the built-in types
  - enables the expressive coding style the community considers Pythonic

- A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users
  - purpose of special methods repr and str exist in the data model
- emulating sequences is one of the most widely used applications of the special methods
- Python offers a rich selection of numeric types, aided by operator overloading, from the builtins to decimal.Decimal and fractions.Fraction, all supporting infix arithmetic operators
- · Python data model used in documentation
  - Python object model used by many authors
- · the Ruby community calls their equivalent of the special methods magic methods.
- the term metaobject protocol is useful to think about the Python data model and similar features in other languages
- · metaobject refers to the objects that are the building blocks of the language itself
- in this context, protocol is a synonym of interface
- metaobject protocol ~= object model an API for core language constructs.
- aspect-oriented programming is much easier to implement in a dynamic language like Python, and several frameworks do it, but the most important is zope.interface

### **II. Data Structures**

# 2. An Array of Sequences

 mastering the standard library sequence types is a prerequisite for writing concise, effective, and idiomatic Python code

#### **Overview of Built-In Sequences**

- mutable
  - list, bytearray, array.array, collections.deque, and memoryview
- immutable
  - tuple, str, and bytes
- flat sequences
  - str, bytes, bytearray, memoryview, and array.array hold items of one type
  - physically store the value of each item within its own memory space, and not as distinct objects
  - more compact, faster, and easier to use

- limited to storing atomic data such as numbers, characters, and bytes
- · container sequences
  - list, tuple, and collections.degue can hold items of different types
  - hold references to the objects they contain, which may be of any type
  - more flexible
  - act surprisingly when they hold mutable objects
  - use with caution with nested data structures

```
Container Iterable Sized
__contains__ __iter__
                              __len__
Sequence
__getitem__
__contains__
__iter__
___reversed___
index
count
MutableSequence
__setitem__
delitem
insert
append
reverse
extend
pop
remove
___iadd__
```

#### **List Comprehensions and Generator Expressions**

- · powerful notations to build and initialize sequences, master them
- listcomps and genexps

```
for symbol in symbols:
    codes.append(ord(symbol))

codes = [ord(symbol) for symbol in symbols]
```

- syntax tip in Python code, line breaks are ignored inside pairs of [], {}, or ()
  - can build multiline lists, listcomps, genexps, dictionaries and the like without using the ugly \ line continuation escape
- in Python 2.x, variables assigned in the for clauses in list comprehensions were set in the surrounding scope
  - the value of outer variables can be overwritten

- prefer listcomps over builtin map and filter
  - may be faster in some cases but not always
  - test
- · to initialize tuples, arrays, and other types of sequences can use listcomp
  - genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor
- use the same syntax as listcomps, but are enclosed in parentheses rather than brackets

```
tuple( ord( symbol) for symbol in symbols)
```

#### **Tuples Are Not Just Immutable Lists**

- · tuples in Python play two roles
  - records with unnamed fields
  - immutable lists
- when a tuple is used as a record, tuple unpacking is the safest, most readable way of getting at the fields
- the new \* syntax makes tuple unpacking even better by making it easier to ignore some fields and to deal with optional fields
- · can use tuple unpacking with any iterable
  - tuple unpacking -> iterable unpacking

```
a, b, *rest = range( 5)
(0, 1, [2, 3, 4])
```

· can use nested tuple unpacking

```
for name, cc, pop, (latitude, longitude) in metro_areas:
```

- · named tuples
  - like tuples, they have very little overhead per instance
  - provide convenient access to the fields by name and a handy .\_asdict() to export the record as an OrderedDict.
- tuples as immutable lists
  - tuple supports all list methods that do not support adding or removing items
    - except reverse but reversed(my\_tuple) works without it
  - add
  - ∘ iadd
  - append
  - clear

- contains
- · copy
- · count
- delitem
- extend
- getitem
- getnewargs
- index
- · insert
- iter
- ∘ len
- mul
- imul
- rmul
- o pop
- · remove
- reverse
- reversed
- ∘ setitem
- sort

#### **Slicing**

- sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize
- multidimensional slicing and ellipsis (...) notation, as used in NumPy, may also be supported by user-defined sequences
- assigning to slices is a very expressive way of editing mutable sequences
- repeated concatenation as in seq \* n is convenient and, with care, can be used to initialize lists of lists containing immutable items

#### Using + and \* with Sequences

#### **Augmented Assignment with Sequences**

- augmented assignment with + = and \*= behaves differently for mutable and immutable sequences
  - \*= these operators necessarily build new sequences
  - if the target sequence is mutable, it is usually changed in place but not always,
     depending on how the sequence is implemented

#### list.sort and the sorted Built-In Functions

- the sort method and the sorted built-in function are easy to use and flexible, thanks to the key optional argument they accept, with a function to calculate the ordering criterion
- · key can also be used with the min and max built-in functions

#### **Managing Ordered Sequences with bisect**

to keep a sorted sequence in order, always insert items into it using bisect.insort; to search it
efficiently, use bisect.bisect

#### When a List Is Not the Answer

- Python standard library provides array.array.
- · NumPy and SciPy are not part of the standard library but should study
- · thread-safe collections.deque
  - comparing its API with that of list in Table 2-3 and mentioning other queue implementations in the standard library

#### 3. Dictionaries and Sets

- · Python dicts are highly optimized
- · hash tables are the engines behind Python's high-performance dicts

#### **Generic Mapping Types**

- · collections.abc
  - Mapping
  - MutableMapping
  - formalize the interfaces of dict
- often extend collections. UserDict instead

Containercontains	Iterable iter	Sized len
^	^	^
Mappinggetitemcontainseqne get items values		

```
MutableMapping
__setitem__
__delitem__
clear
pop
popitem
setdefault
update
```

- · keys must be hashable
  - an object is hashable if it has a hash value which never changes during its lifetime (it needs a **hash** () method), and can be compared to other objects (it needs an **eq** () method)
  - hashable objects which compare equal must have the same hash value
  - the atomic immutable types (str, bytes, numeric types) are all hashable
  - · a frozenset is always hashable, because its elements must be hashable by definition
  - a tuple is hashable only if all its items are hashable

#### dict Comprehensions

· dictcomp added in Python 2.7

```
country_code = {country: code for code, country in DIAL_CODES}
```

#### **Overview of Common Mapping Methods**

- defaultdict and OrderedDict in collections
  - · clear
  - contains
  - copy
  - · copy
  - default\_factory
  - delitem
  - fromkeys
  - get
  - getitem
  - items
  - iter
  - keys
  - len
  - · missing

- move\_to\_end
- pop
- popitem
- reversed
- setdefault
- setitem
- update
- values

#### Handling Missing Keys with setdefault

- dict access with d[k] raises an error when k is not an existing key
- d.get(k, default) is an alternative to d[k] whenever a default value is more convenient than handling KeyError
- when updating the value found (if it is mutable), using either getitem or get is awkward and inefficient

```
my_dict.setdefault(key, []).append(new_value)

# same as
if key not in my_dict:
    my_dict[key] = []
my_dict[ key]. append( new_value)
```

 setdefault uses a single lookup whereas the second example uses at least two and three if its not found

### **Mappings with Flexible Key Lookup**