Python Data Structures

Fabrice Rossi

CEREMADE
Université Paris Dauphine

2020

Outline

Lists

Tuples

Sets

Dictionaries

NumPy

Common concepts

Something-able objects

- Python objects share some capabilities, such as being iterable
- high level operations available on data structures are linked to those capabilities
- important categories:
  - *iterable*: can be used in `for` loops
  - *subscriptable:*
    - can be used with the `[]` operator
    - *indexable*: `[k]` where `k` is an integer
    - *sliceable*: `[b:e:t]`
  - *mutable*: objects that can be modified (as opposed to *immutable*)
  - *callable*: can be called (e.g. functions)
  - *hashable*: to be discussed

Outline

Lists

Grouping values

- a python `list` is a group of values arranged in a certain order
- literal value `[val1, val2, ..., valn]`
- mixed types are accepted

Example

The program

```
my_list = [1, 2, -5]
my_other_list = [True, 'foo', 2.5]
print(my_list)
print(my_other_list)
```

prints

```
[1, 2, -5]
[True, 'foo', 2.5]
```

Core aspects

Global operations

- literal values `[ comma separated values]`
- empty list `[]`
- length: `len` function
- concatenation of 2 lists: `l1 + l2` returns a list made of the elements of l1 followed by the elements of l2, e.g.
  `[1, True, 'bob'] + [2, False]` is
  `[1, True, 'bob', 2, False]`
- replication of 1 list: `l1 * nb` returns a list made of l1 concatenated with itself nb times, e.g.
  `[1.5, False] * 3` is
  `[1.5, False, 1.5, False, 1.5, False]`

Core aspects

Example

The program

```
print(4 * [-1, 2])
x = 2 * [0, 1] + [True, False]
print(len(x))
print(x)
```

prints

```
[-1, 2, -1, 2, -1, 2, -1, 2]
6
[0, 1, 0, 1, True, False]
```

Content access

Iterable

- lists are **iterable**
- the program

```
x = [True, 'abcd', 2, -1.4, 4]
for t in x:
    print(t)
```

prints

```
True
abcd
2
-1.4
4
```

Indexable

- lists are **indexable**
- values are numbered from 0 to `len(list)-1`
- bracket based access `list[k]`
- the program

```
x = [True, 'abcd', 2, -1.4, 4]
print(x[1])
for k in range(2,len(x)):
    print(k)
```

prints

```
abcd
2
3
4
```

Changing content

List modification

- lists are **mutable**: a given list object may be modified
- several possibilities:
  - changing the value at a given position with `list[k]=t`
  - modifying the *structure* of the list via *methods*
    - appending elements
    - inserting elements
    - removing elements
  - reorganizing the content (sorting)
- complex consequences: the object is modified!

Content modification

Example

The program

```
x = [2, 4, -1, 'abcd', 'foo']
y = x
x[0] = 3
print(y)
y[1] = True
print(x)
```

prints

```
[3, 4, -1, 'abcd', 'foo']
[3, True, -1, 'abcd', 'foo']
```

Interpretation

- `x[0] = 3` sets the first object referenced by the list `x` to 3
- `x` and `y` are two names for a single object
- some operations modify the content of the object
- modifications made via `x` are visible through `y` (and vice versa)

Methods

Element level modifications

- `l`: a list, `x`: an object, `i`: an integer
- `l.append(x)`: adds `x` at the end of `l`
- `l.insert(i,x)`: inserts `x` at position `i` in `l`
- `l.pop(i)`: removes the element at position `i` and returns it
- `l.remove(x)`: removes the first instance of `x` in `l`

Global level modifications

- `l.sort()`: sorts `l`
- `l.reverse()`: puts `l` in the reserve of the original order
- `l.clear()`: empties `l`

Example

Code

```
my_list = []
for k in range(5):
    my_list.append(k//2)
    if k%2==0:
        my_list.append('a' * k)
print(my_list)
my_list.insert(1, 4)
print(my_list)
print(my_list.pop(3))
print(my_list)
my_list.remove(1)
print(my_list)
my_list.reverse()
print(my_list)
my_list.clear()
print(my_list)
```

Result

```
[0, '', 0, 1, 'aa', 1, 2, 'aaaa']
[0, 4, '', 0, 1, 'aa', 1, 2, 'aaaa']
0
[0, 4, '', 1, 'aa', 1, 2, 'aaaa']
[0, 4, '', 'aa', 1, 2, 'aaaa']
['aaaa', 2, 1, 'aa', '', 4, 0]
[]
```

Methods and functions

Search and count

- `l`: a list, `x`: an object
- `l.count(x)`: returns the number of times `x` occurs in `l`
- `l.index(x)`: returns the first position of `x` in `l` (raises an exception if `x` does not appear in `l`)

Summary functions (work on any iterable)

- `max(l)`
- `min(l)`
- `sum(l)`
- `any(l)`: **`True`** is at least one element of `l` is **`True`**
- `all(l)`: **`False`** is at least one element of `l` is **`False`**

Operators and statement

Membership

- `in` and `not in` are operators which test whether an object is a member of another object
- apply to many types, including all iterable objects
- `l`: a list, `x`: an object
- `x in l`: evaluates to `True` if `x` appears in `l` and to `False` if it does not
- `x not in l` is the negation of `x in l`

Deletion

- the `del` instruction can be used to delete something
- if `l` is a list, `del l[k]` removes the element of position `k` from the list, shifting all the rest

Slices

Sliceable

- lists are **sliceables**
- standard read semantics
  - general form `l[first:last:step]`
  - exactly the same interpretation as in strings
  - new independent list with content taken in `l`
- write semantics
  - `l[first:last:step] = a_list`
  - replaces the slice by the new list, regardless of their respective sizes

Example

```
my_list = [1, 2, 3, 4]
print(my_list[1:])
print(my_list[::-1])
my_list[1:3] = [5]
print(my_list)
my_list[1:] = []
print(my_list)
```

Result

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

Comprehensions

Efficient list construction

- comprehensions create lists in a concise way from another iterable object
- general form `[expression for var in iterable]`
  - any `iterable` can be used
  - `expression` is generally based on the `var` variable
- semantics
  - for each value in the `iterable`
    - `var` is set to the value
    - `expression` is evaluated
  - the result is the list of all the values of `expression`

Examples

Comprehension

```
ml = [expression for var in iterable]
```

Equivalent code

```
ml = []
for var in iterable:
    ml.append(expression)
```

Program

```
print([x for x in 'ABCD'])
ml = [x**2 for x in range(5)]
print(ml)
print([2 * x - 1 for x in ml])
```

Output

```
['A', 'B', 'C', 'D']
[0, 1, 4, 9, 16]
[-1, 1, 7, 17, 31]
```

Filtered comprehensions

Comprehensions can be filtered to include only some of the values of the expression/variable

Comprehension

```
ml = [expression for var in iterable
                         if condition]
```

Equivalent code

```
ml = []
for var in iterable:
    if condition:
        ml.append(expression)
```

Program

```
print([ x + 1 for x in range(6)
                    if x%2 == 1])
print([ a for a in 'AbCdDe' if a.isupper()])
print([ 2 * y + 1 for y in [1, -2, 4, 5]
                    if y >= 0 ])
```

Output

```
[2, 4, 6]
['A', 'C', 'D']
[3, 9, 11]
```

Nested comprehensions

Comprehensions can be created using nested loops

Comprehension

```
ml = [expression for x in x_iterable
                 for y in y_iterable]
```

Equivalent code

```
ml = []
for x in x_iterable:
    for y in y_iterable:
        ml.append(expression)
```

Program

```
print([x + 2* y for x in range(3)
                for y in range(1,4)])
print([2* x - y for x in range(3)
                for y in range(x + 1)])
```

Output

```
[2, 4, 6, 3, 5, 7, 4, 6, 8]
[0, 2, 1, 4, 3, 2]
```

Nested loops and conditions can be combined arbitrarily

Outline

Tuples

Grouping values

- a python `tuple` is a group of values arranged in a certain order
- literal value `(val1, val2, ..., valn)`
- mixed types are accepted

Tuples

Grouping values

- a python `tuple` is a group of values arranged in a certain order
- literal value `(val1, val2, ..., valn)`
- mixed types are accepted
- Isn't that a list?

Tuples

Grouping values

- a python `tuple` is a group of values arranged in a certain order
- literal value `(val1, val2, ..., valn)`
- mixed types are accepted
- Isn't that a list?

Immutable

- once created a `tuple` cannot change (neither globally, nor locally)
- a type of immutable object
- tuples can be seen as immutable lists

Tuple values

Literal values

- parenthesis are optional for tuples with at least two elements
- `()` is the empty tuple
- `tuple()` creates also the empty tuple
- for a tuple with only one element, a comma is mandatory: `(1,)` or `5,` are such tuples

Example

```
x = 1, 2, 3
print(x)
y = (2) # not a tuple
print(y)
z = 1,
print(z)
```

Output

```
(1, 2, 3)
2
(1,)
```

Working with tuples

Supported operations

- tuples are **iterable**, **indexable** and **sliceable**
- they support the *common sequence operations*
  - `in` and `not in` statements
  - `+` and `*` operators
  - `len`, `min` and `max` functions
  - `index` and `count` methods
- as well as the standard iterable functions `sum`, `any` and `all`
- tuples can be created from any iterable using `tuple(iterable)`

Example

Program

```
x = tuple(range(4))
y = (2,3) * 3
print(x)
print(y)
for k in x:
    print(k)
    print(k in y)
print(sum(y))
for i in range(0,len(y),2):
    print(i, y[i])
```

Output

```
(0, 1, 2, 3)
(2, 3, 2, 3, 2, 3)
0
False
1
False
2
True
3
True
15
0 2
2 2
4 2
```

Unpacking

Sequence/iterable unpacking

- Python supports ways to *unpack* a sequence into several variables
- general format

```
var1, var2, ..., varn = sequence/iterable
```

- constraint: exactly the same number of variables as there are elements in the sequence/iterable
- `var1` names the first value of the sequence, `var2` the second, and so on

Example

```
x, y, z = [2, 3, 4]
print(x)
print(y)
print(z)
```

Result

```
2
3
4
```

Packing

Tuple packing

- creating a tuple can be seen as *tuple packing*, a.k.a. assembling values into a tuple
- e.g. in `x = 1, -2, 4` the values `1`, `-2` and `4` are *packed* in a tuple `(1, -2, 4)`

Implictly use with unpacking

Example

```
a = 2
b = 3
a, b = b, a
print(a)
print(b)
```

Result

```
3
2
```

Packing

Tuple packing

- creating a tuple can be seen as *tuple packing*, a.k.a. assembling values into a tuple
- e.g. in `x = 1, -2, 4` the values `1`, `-2` and `4` are *packed* in a tuple `(1, -2, 4)`

Implictly use with unpacking

Example

```
a = 2
b = 3
a, b = b, a
print(a)
print(b)
```

Explanation

- `b, a` is a tuple packing
- equivalent to `(b, a)`
- the tuple resulting tuple is `(3, 2)`
- `(3, 2)` is unpacked in `a` and `b`

Multiple values in one return

Packing results

- tuple packing is very useful to return several values in a single return instruction
- sequence unpacking can be used to split the result into several variables

Example

```
def plusminus(x, y):
    return x + y, x - y

t = plusminus(6, 1)
print(t)
a, b = plusminus(4, 3)
print(a)
print(b)
```

Result

```
(7, 5)
7
1
```

Enumerating an iterable

`enumerate`

- the built in `enumerate` function constructs a new iterable object from another one
- when one iterates using this new object, each run of the loop is given both the index in the original iterable and the corresponding value, packed into a tuple
- unpacking is convenient here

Example

```
foo = [1, -2, True, 'abcd']
for t in enumerate(foo):
    print(t)
```

Result

```
(0, 1)
(1, -2)
(2, True)
(3, 'abcd')
```

Enumerating an iterable

`enumerate`

- the built in `enumerate` function constructs a new iterable object from another one
- when one iterates using this new object, each run of the loop is given both the index in the original iterable and the corresponding value, packed into a tuple
- unpacking is convenient here

Example

```
foo = [1, -2, True, 'abcd']
for index, value in enumerate(foo):
    print(index, value)
```

Result

```
0 1
1 -2
2 True
3 abcd
```

Function calls

Packing parameters

- a function call needs as many arguments as there are parameters (up to default parameters)
- the arguments can be unpacked from a tuple
- specific unpacking operator `*`
- unpacking must be explicit

Example

```
def foo(a, b):
    return a + b

x = 2, 3
print(foo(*x))
```

Result

```
5
```

Outline

Hashing function

Fixed size representation

- a hashing function maps objects of arbitrary size to *hash values* of a fixed size (typical a n bit integer)
- hash values are also named *hash codes*, *digests* or *hashes*
- basic example
  - division hashing
  - the binary representation of an object o is interpreted as a (possibly very large) integer $i(o)$
  - $h(o) = i(o) \bmod 2^n$ (remainder of the Euclidean division)

Applications

- Many!
- Hash tables
- signature and other cryptographic application

Hashable objects

`hash`

- built-in python function
- gives the hash code of any **hashable** object
  - must have a constant hash code once created
  - must be comparable to other objects (with hash consistency)
- in general
  - immutable base objects are hashable
  - mutable objects are not hashable
  - immutable containers such as tuples are hashable if their content is

Sets

Representing sets

- mathematical sets are represented by python `set`
- a set contains only hashable objects
- literal value `{val1, ..., valn}` with `set()` for an empty set
- `{}` is not an empty set

Use

- sets are fast structures: testing whether an object belongs to a set takes on average a constant time regardless of the size of the set
- useful for computing e.g. the set of words used in a document

Working with sets

Supported operations

- sets are **iterable** but they are neither indexable nor sliceable
- sets support some of the sequence operations (but they are not sequences!)
  - `in` and `not in` statements
  - `len`, `min` and `max` functions
- as well as the standard iterable functions `sum`, `any` and `all`

Set operations

- `|`, `&`, `^`, `-`: respectively union, intersection, symmetric difference and difference
- `<`, `<=`, `==`, `>=`, `>`: testing for inclusion

Example

Program

```
A = {'a', 'b', 'c'}
B = {'c', 'd', 'e', 'f'}
print(A | B) # union
print(A & B) # intersection
print(B - A) # in B but not in A
print(A ^ B) # symmetric difference
print(len(A)) # cardinal
print('d' in A) # belongs to
print({'b', 'c'} <= A) # subset
for x in B:
    print(x)
```

Output

```
{'c', 'a', 'e', 'd', 'b', 'f'}
{'c'}
{'e', 'd', 'f'}
{'e', 'a', 'b', 'd', 'f'}
3
False
True
c
e
d
f
```

Modifiying sets

Mutable sets

- sets are **mutable**
- they provide element oriented modification methods:
  - `s` a `set` and `x` any hashable object
  - `s.add(x)`: adds `x` to `s`
  - `s.remove(x)`: removes `x` from `s` if `x in s` and raises a `KeyError` exception if `x not in s`
  - `s.discard(x)`: removes `x` from `s` if `x in s` (no error in the other case)
  - `s.pop()`: removes and returns an arbitrary element from `s`
- `s.clear()`: removes everything from `s`
- sets can also be updated using other sets with operations of the form `s op= t` where `op` is one of the set operators. For instance `s |= t` updates `s` to the union of `s` and `t`

Example

Program

```
A = set()
for x in range(5):
    A.add((x + 1) // 2)
print(A)
B = A
A.discard(1)
print(B)
for t in 'abcdef':
    A.add(t)
print(A)
A -= {'a', 'b', 'c'}
print(B)
```

Output

```
{0, 1, 2}
{0, 2}
{0, 2, 'b', 'e', 'f', 'c', 'a', 'd'}
{0, 2, 'e', 'f', 'd'}
```

Example

Program

```
A = set()
for x in range(5):
    A.add((x + 1) // 2)
print(A)
B = A
A.discard(1)
print(B)
for t in 'abcdef':
    A.add(t)
print(A)
A -= {'a', 'b', 'c'}
print(B)
```

Output

```
{0, 1, 2}
{0, 2}
{0, 2, 'b', 'e', 'f', 'c', 'a', 'd'}
{0, 2, 'e', 'f', 'd'}
```

Notice the reference effects!

Immutable Sets

Sets in sets

- Sets are mutable and thus not hashable
- One cannot put a set in a set. The program

```
A = { {1}, {3, 4}}
```

fails with an error

```
  File "setinset.py", line 1, in <module>
    A = { {1}, {3, 4} }

TypeError: unhashable type: 'set'
```

- Notice this is also true for e.g. lists

`frozenset`

- A `frozenset` is a set that cannot be modified
- it supports all the non modification methods of `set`
- created via `frozenset(iterable)`
- **hashable**

Example

Program

```
A = { 1, 2 }
B = { 3, 4 }
C = [x for x in range(3)]
D = {frozenset(A),
     frozenset(B),
     tuple(C)}
print(D)
A.add(3)
B.add(5)
C.append(A)
print(A, B, C)
print(D)
```

Output

```
{frozenset({3, 4}), (0, 1, 2),↵
  frozenset({1, 2})}
{1, 2, 3} {3, 4, 5} [0, 1, 2, {1, 2, 3}]
{frozenset({3, 4}), (0, 1, 2),↵
  frozenset({1, 2})}
```

Example

Program

```
A = { 1, 2 }
B = { 3, 4 }
C = [x for x in range(3)]
D = {frozenset(A),
     frozenset(B),
     tuple(C)}
print(D)
A.add(3)
B.add(5)
C.append(A)
print(A, B, C)
print(D)
```

Output

```
{frozenset({3, 4}), (0, 1, 2),↵
  frozenset({1, 2})}
{1, 2, 3} {3, 4, 5} [0, 1, 2, {1, 2, 3}]
{frozenset({3, 4}), (0, 1, 2),↵
  frozenset({1, 2})}
```

Immutable

- `set` → `frozenset`
- `list` → `tuple`

Set comprehensions

Creating sets

- comprehensions can be used to create sets
- similar syntax to the one of list comprehension but with curly braces

```
{ expression for variable in iterable }
```

- exactly the same principles as for lists except for the hashability constraint

Example

Program

```
print({x % 2 for x in range(5)})
print( {x + y for x in range(5)
               for y in range(x+1)
                if (x-y) %2==0})
print( { tuple([y**x for y in range(1,4)])
         for x in range(1,4)})
```

Output

```
{0, 1}
{0, 2, 4, 6, 8}
{(1, 8, 27), (1, 4, 9), (1, 2, 3)}
```

Example

Program

```
print({x % 2 for x in range(5)})
print( {x + y for x in range(5)
                for y in range(x+1)
                 if (x-y) %2==0})
print( { tuple([y**x for y in range(1,4)])
          for x in range(1,4)})
```

Output

```
{0, 1}
{0, 2, 4, 6, 8}
{(1, 8, 27), (1, 4, 9), (1, 2, 3)}
```

Immutable

Notice the use of `tuple` to circumvent list mutability

Outline

Lists

Tuples

Sets

Dictionaries

NumPy

Dictionaries

Principle

- lists are limited by the integer based indexing scheme
- dictionaries replace integers by arbitrary keys
  - a dictionary contains (key, value) pairs
  - keys are unique (they form a set)
  - values are accessed through keys

Literal values

- `{}` is an empty dictionary (also with `dict()`)
- general form

```
{ key1: value1, key2: value2, ...}
```

- keys must be hashable

Accessing a dictionary

Keys as indices

- dictionaries are **indexable**
- if `d` is a `dict`, `d[key]` returns the *value* associated to the `key`
- raises a **`KeyError`** is the `key` does not exist in `d`
- `d.get(key)` returns either the value associated to `key` or **`None`** if the `key` is not in the dictionary

Example

```
D = {1: 'foo',
     'bar': 'tt',
     (3, 4): 5}
print(D[1])
print(D.get((3,4)))
print(D.get('foo'))
print(D['bar'])
```

Output

```
foo
5
None
tt
```

Note

- keys are completely arbitrary *hashable* values
- values and keys are completely separated

Using dictionaries

Supported operations

- dictionaries are **iterable**
  - default iteration operates on *keys*
  - one can also iterate over values and (key, value) pairs
- dictionaries support some default operations **all interpreted on the keys**
  - `in` and `not in`
  - `len`, `min` and `max`
  - `sum`, `any` and `all`
- notice that most of those operations do not make much sense on dictionaries as generally keys are not numeric!

Views and Iteration

Dictionary views

- dictionaries provide dynamic *views* of their content:
  - `d.keys()` a “set” of the keys in `d`
  - `d.values()` a “list” of the values in `d`
  - `d.item()` a “collection” of the (key, value) pairs in `d`
- the views support a minimal set of operations
  - they are **iterable**
  - `len` is supported
  - the `in` operator is supported
- any modification of the dictionary is reflected in all its views

Example

Code

```
D = {'a': 1, 'b': -3,
     'c': 7, 'd': 4}
print(len(D))
print(max(D))
print('e' in D, 4 in D)
for k in D:
    print(k)
for val in D.values():
    print(val)
for k, val in D.items():
    print(k,'->',val)
print(('b', -3) in D.items())
```

Output

```
4
d
False False
a
b
c
d
1
-3
7
4
a -> 1
b -> -3
c -> 7
d -> 4
True
```

Modifying dictionaries

Mutable dictionaries

- dictionaries are **mutable**
- some element oriented operations:
  - `d` a `dict`, `k` a hashable key, `v` any value
  - `d[k] = v` either inserts a new key `k` with the `v` value or update the value associated to `k` to `v`
  - `del d[k]` removes `k` from `d` (raises `KeyError` if `k` is not in `d`)
  - `d.pop(k)` returns the value associated to `k` and removes the pair from `d` (raises `KeyError`)
  - `d.pop(k, val)` behaves as `d.pop(k)` if `k` is in `d` and return `val` is this is not the case
- `d.clear()` removes everything from `d`
- notice Python does not provide a *frozen dictionary*

Example

Code

```
D = {} # empty
for k in range(5):
    D[k] = k**2
print(D)
H = D
del H[2]
H[5] = 5**2
print(D.pop(6, None))
print(D)
```

Output

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
None
{0: 0, 1: 1, 3: 9, 4: 16, 5: 25}
```

Example

Code

```
D = {} # empty
for k in range(5):
    D[k] = k**2
print(D)
H = D
del H[2]
H[5] = 5**2
print(D.pop(6, None))
print(D)
```

Output

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
None
{0: 0, 1: 1, 3: 9, 4: 16, 5: 25}
```

Notice the reference effects!

Creating dictionaries

Dictionary comprehension

- comprehensions can be used to create dictionaries
- similar syntax to the one of list comprehension but with curly braces

```
{ key expr: value expr for variable in iterable }
```

- exactly the same principles as for lists except for the hashability constraint

Example

Code

```
print({k: k**2 for k in range(4)})
print({k**2: k for k in range(1, 5)})
D = {k: { l for l in range(3, k)
              if k%l==0}
        for k in range(9, 30, 3)}
for k,v in D.items():
    print(k,'->',v)
```

Output

```
{0: 0, 1: 1, 2: 4, 3: 9}
{1: 1, 4: 2, 9: 3, 16: 4}
9 -> {3}
12 -> {3, 4, 6}
15 -> {3, 5}
18 -> {9, 3, 6}
21 -> {3, 7}
24 -> {3, 4, 6, 8, 12}
27 -> {9, 3}
```

Creating dictionaries

Zipping iterables

- `zip(iter1, iter2)` returns a new iterable object whoe content is made of pairs obtained by iterating `iter1` and `iter2` simultaneously
- elements of `zip(...)` are tuples
- works with more than two iterators
- `dict(zip(iter1,iter2))` creates a dictionary with elements of `iter1` as keys and elements of `iter2` as values (in their respective order)

Example

Code

```
k = list('abcdefgh')
v = [x ** 2 for x in range(1, 9)]
D = dict(zip(k, v))
for x, y in D.items():
    print(x, '->', y)
```

Output

```
a -> 1
b -> 4
c -> 9
d -> 16
e -> 25
f -> 36
g -> 49
h -> 64
```

Dictionary unpacking

Function calls

- positional matching for function calls: tuple unpacking
- keyword matching for function calls: dictionary unpacking!
- principle:
  - a function with parameters `f(a,b,c,...)`
  - a dictionary `d` with the parameters as keys
  - unpacking calls: `f(**d)`
  - equivalent to `f(a=d[a],b=d[b],c=d[c],...)`

Example

```
def f(a, b):
    return a**b

D = {}
D['a'] = 2
D['b'] = 3
print(f(**D))
```

Output

```
8
```

Outline

NumPy

Rationale

- built-in Python structures have some limitations
  - memory usage is large
  - mathematical operations can be slow
  - standard numerical algorithms are not supported
- NumPy solves those problems
  - memory efficient
  - fast
  - feature rich

Arrays

Main type

- NumPy's main type is `ndarray`
- it represents a multidimensional array
  - uniform content (e.g. real numbers)
  - dimensions are called axis in NumPy
  - one dimension : vector
  - two dimensions : matrix
  - more dimensions : tensor

Example

```
import numpy as np
x = np.array([1, 2, 3])
print(x)
y = np.array([[3, 4], [-1, 6]])
print(y)
```

Output

```
[1 2 3]
[[ 3  4]
 [-1  6]]
```

Attributes

Main attributes

- `ndim`: number of axis
- `shape`: size of the array along each axis
- `size`: number of values in the array
- `dtype`: type of the elements

Example

```
import numpy as np
x = np.array([1, 2, 3])
print(x.ndim, x.shape)
print(x.size, x.dtype)
y = np.array([[3.5, 4.1], [-1.25, 6]])
print(y.ndim, y.shape)
print(y.size, y.dtype)
```

Output

```
1 (3,)
3 int64
2 (2, 2)
4 float64
```

Creating arrays

Creation functions

- `array`: from a `list` or nested lists
- filling with constants:
  - `ones` and `zeros`
  - `full`
- random values
  `random.random`
- vector ranges:
  - `arange` (similar to `range`)
  - `linspace`
- matrices:
  - `eye` (identity)
  - `diag`

Example

```
import numpy as np
print(np.ones((2,2)))
print(np.zeros((2,3)))
print(np.full((2,), 4))
print(np.random.random((3,)))
print(np.linspace(0, 1, 5))
print(np.eye(2))
print(np.diag(np.arange(0, 3, 1)))
```

Output

```
[[1. 1.]
 [1. 1.]]
[[0. 0. 0.]
 [0. 0. 0.]]
[4 4]
[0.94470698 0.77223689 0.4571132 ]
[0.   0.25 0.5  0.75 1.  ]
[[1. 0.]
 [0. 1.]]
[[0 0 0]
 [0 1 0]
 [0 0 2]]
```

Calculation

Elementwise operations

- standard arithmetic operators
  - between two `ndarray` (off the same shapes)
  - between a single value and a `ndarray`
- boolean operations
  - comparison operators
  - boolean operators & and |
  - avoid using **`and`** and **`or`**

Example

```
import numpy as np
x = np.ones(4)
y = np.linspace(0, 3, 4)
print(x + y)
print(x * y)
print((2 * x) ** y)
print((x - 1) < y)
print((x > y) | (y > 1))
```

Output

```
[1. 2. 3. 4.]
[0. 1. 2. 3.]
[1. 2. 4. 8.]
[False  True  True  True]
[ True False  True  True]
```

In-place calculation

Modifying arrays

- for efficiency reasons
- especially with large arrays
- modifications are done by some specific methods, e.g. reshaping methods
- in-place operators (e.g. `+=`) modify the calling object
- copies are obtained via the `copy` method

Example

```
import numpy as np
x = np.linspace(-1, 1, 5)
print(x)
y = x
z = x.copy()
x *= 2
print(x)
print(y)
print(z)
```

Output

```
[-1.  -0.5  0.   0.5  1. ]
[-2. -1.  0.  1.  2.]
[-2. -1.  0.  1.  2.]
[-1.  -0.5  0.   0.5  1. ]
```

Calculation

Universal functions (`ufunc`)

- functions that operate element by element on arrays
- standardized options and behaviors
- backend for operators (e.g. `+` is `add`)

Example

```
import numpy as np
x = np.linspace(-1, 1, 3)
y = np.linspace(0, 2, 3)
print(x, y)
w = np.add(x, y)
z = np.exp(w)
print(w)
print(z)
print(np.rint(z))
print(np.maximum(x, y))
```

Output

```
[-1.  0.  1.] [0. 1. 2.]
[-1.  1.  3.]
[ 0.36787944  2.71828183 20.08553692]
[ 0.  3. 20.]
[0. 1. 2.]
```

Aggregations

(Semi)Global operations

- `numpy` includes global operations (as opposed to element by element ones)
- some examples (methods)
  - `sum` and `prod`
  - `min` and `max`
  - `mean`, `std` and `var`
- function examples
  - `median`
  - `quantile` and `percentile`

Axes

- all of those methods/functions accept an optional `axis` parameter
- axis refers to a dimension of the array, the one over which the aggregation is carried
- dimensions are numbered starting from 0

Aggregations

```
import numpy as np
x = np.linspace(0, 2, 5)
print(x)
print(x.sum())
print(np.median(x))
A = np.array([[1, 2],[3, 4]])
print(A)
print(A.max(axis=0))
print(A.sum(axis=1))
C = np.array([[[1, 2],[3, 4]],
              [[1, 2],[3, 4]]])
print(C)
print(C.sum(axis=2))
print(C.sum(axis=1))
print(C.sum(axis=0))
```

```
[0.  0.5 1.  1.5 2. ]
5.0
1.0
[[1 2]
 [3 4]]
[3 4]
[3 7]
[[[1 2]
  [3 4]]

 [[1 2]
  [3 4]]]
[[3 7]
 [3 7]]
[[4 6]
 [4 6]]
[[2 4]
 [6 8]]
```

Vectors and matrices

Vector space operations

- summing vectors or matrices via `+` or `add`
- multiplying by a scalar via `*` or `multiply`

Products

- inner product using `dot`
- matrix/matrix and matrix/vector product via `@` or `matmul`
- norm via `linalg.norm`

Example

```
import numpy as np
import math as m
x = np.array([1, 2, 3])
A = np.array([[1,0.2,0],
              [0.2,1,0.1],
              [0,0.1,1]])
print(np.dot(x, x))
print(m.sqrt(np.dot(x, x)))
print(np.linalg.norm(x))
print(A@x)
```

Output

```
14
3.7416573867739413
3.7416573867739413
[1.4 2.5 3.2]
```

Application example

Power method

- A a diagonalizable matrix
- we search for its dominant eigenvalue λ
- can be obtained via the power method
- it iterates $x_{k+1} = \frac{1}{\|Ax_k\|} Ax_k$ until convergence

function POWER(A, ϵ)
  choose a random vector x
  repeat
    $x' \leftarrow \frac{1}{\|Ax\|} Ax$
    $\delta \leftarrow \|x' - x\|$
    $x \leftarrow x'$
  until $\delta \leq \epsilon$
  $\lambda = \frac{x^T Ax}{x^T x}$
  return (λ, x)
end function

Application example

Implementation

```
import numpy as np

def power_method(A, prec=1e-8):
    x = np.random.random(A.shape[1])
    iterate = True
    while(iterate):
        nx = A@x
        nx /= np.linalg.norm(nx)
        delta = np.linalg.norm(x - nx)
        x = nx
        iterate = delta > prec
    eigenvalue = (np.dot(x, A@x))/(np.dot(x, x))
    return eigenvalue,x

A = np.array([[1,0.2,0],
              [0.2,1,0.1],
              [0,0.1,1]])
result = power_method(A)
print(result[0])
print(result[1])
```

Result

```
1.2236067977499787
[0.63245555 0.70710678 0.31622773]
```

Linear algebra

Numerous high level operations

- most of them in the submobule `numpy.linalg`
- matrix decompositions
  - `cholesky`
  - `qr`
- eigenvalues `eig`
- singular values `svd`
- determinant `det`
- equation solution and inversion `solve`, `inv`

Broadcasting

Adapting shapes

- arrays with different shapes cannot be combined directly
- a typical case: `np.array([1, 2]) + 2`
- strict interpretation is limiting
  - e.g. `2 * np.array([1, 2])` would be rejected
  - matrix/vector combinations would be harder to implement
- *broadcasting* to the rescue
  - key idea: interpret low dimensional arrays as high dimensional ones with a size one on each “missing” dimension
  - replicate the values on this missing dimension

Broadcasting

Principles

- stretching
  - elements are copied along a dimension
  - but only when the size of dimension is one
- extending
  - arrays with k dimensions can be extended into $k + p$ dimensions
  - the sizes of the additional dimensions are one
  - new dimensions come “in front” of the others

Examples

- a scalar can be broadcasted into a constant array of any shape
- $(1, 2, 3)$ can be broadcasted into

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

but **not** into

$$\begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{pmatrix}$$

Examples

```
import numpy as np
x = np.array([1, 2, 3])
y = x + 1 # 1 -> array([1, 1, 1])
print(y)
A = np.diag(x)
print(A)
B = A + y # y -> array([y, y, y])
print(B)
C = A - 1 # -1 -> full((3,3),-1)
print(C)
u = np.array([[1, 2]])
v = np.array([[1], [2]])
# double broadcasting
D = u + v
print(D)
```

```
[2 3 4]
[[1 0 0]
 [0 2 0]
 [0 0 3]]
[[3 3 4]
 [2 5 4]
 [2 3 7]]
[[ 0 -1 -1]
 [-1  1 -1]
 [-1 -1  2]]
[[2 3]
 [3 4]]
```

Example: normalizing a matrix

```
import numpy as np
A = np.random.random((10,3))
# inplace calculation
A -= A.mean(axis=0)
A /= A.std(axis=0)
print(A.mean(axis=0))
print(A.std(axis=0))
```

```
[ 7.10542736e-16 -1.88737914e-16↵
  4.44089210e-17]
[1. 1. 1.]
```

Reshaping

Principle

- modifying the shape without changing the data
- typical example: transposition (`T` attribute)
- general case
  - `reshape` function/method
  - the array is read in a given order
  - and written in a similar order into a new shape
- default order: last axis varies faster

Example

```
import numpy as np
A = np.random.random((2,3))
A = A.round(decimals=2)
print(A)
print(A.T)
print(A.reshape((3,2)))
```

Output

```
[[0.56 0.06 0.07]
 [0.54 0.04 0.53]]
[[0.56 0.54]
 [0.06 0.04]
 [0.07 0.53]]
[[0.56 0.06]
 [0.07 0.54]
 [0.04 0.53]]
```

Stacking

Combining arrays

- “gluing” arrays
- simple semantics for up to 3 dimensions
  - `vstack`: concatenation on the first axis
  - `hstack`: concatenation on the second axis
- `stack`: concatenation on a new axis
- `concatenate`: concatenation on a given axis

Example

```
import numpy as np
x = np.array([1, 2, 3])
A = np.vstack((x,x))
print(A)
B = np.stack([x,x],1)
print(B)
print(np.hstack((x,x)))
```

Output

```
[[1 2 3]
 [1 2 3]]
[[1 1]
 [2 2]
 [3 3]]
[1 2 3 1 2 3]
```

Indexing

Direct access

- accessing directly to the content of an array is generally not a good idea for efficiency reasons
- supported by indexing (and slicing) facilities
- `ndarrays` are **indexable** and **sliceable**
- multidimensional extension
  - missing dimensions use `:`
  - ellipsis `...`

Example

```
import numpy as np
A = np.random.random((2,3))
A = A.round(decimals=2)
print(A)
print(A[1]) # A[1,:]
print(A[:,2])
print(A[1,1:3])
```

Output

```
[[1.   0.06 0.83]
 [0.97 0.28 0.76]]
[0.97 0.28 0.76]
[0.83 0.76]
[0.28 0.76]
```

Advanced indexing

Indexing with arrays

- arrays can be used to index other arrays
- two modes
  - one indexing array of integers per dimension to index
  - or one boolean array of the same shape as the index array
- many subtleties

Example

```
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
print(A)
b = np.array([0, 2])
c = np.array([0, 1])
print(A[b,c])
B = np.array([[1, 1], [0,2 ]])
C = np.array([1, 0])
print(A[B, C])
print(A[A>=3])
```

Output

```
[[1 2]
 [3 4]
 [5 6]]
[1 6]
[[4 3]
 [2 5]]
[3 4 5 6]
```

Views

Copies versus views

- a view is an array that shares its data with another array: modifying one changes the other
- pro: saves memory and processing time
- con: aliases
- views are created
  - explicitly with the `view` method
  - by slicing
  - by some reshaping operations
- a view has a `base` attribute which contains its original array
- the `shares_memory` function can be used to test whether to arrays are related

Views

```
import numpy as np
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
B = A[0:2,:]
print(B)
print(A.base is None)
print(B.base is A)
B[1,1] = 0
print(A)
C = A.T
x = C[0,:]
print(x)
x[1] = -1
print(A)
print(B)
print(C)
```

```
[[1 2]
 [3 4]]
True
True
[[1 2]
 [3 0]
 [5 6]]
[1 3 5]
[[ 1  2]
 [-1  0]
 [ 5  6]]
[[ 1  2]
 [-1  0]]
[[ 1 -1  5]
 [ 2  0  6]]
```

Iterating over an array

- is generally not very useful
- but can be done
  - `ndarrays` are **iterable** (over the first dimension)
  - the `flat` attribute is an element level iterator
  - the `nditer` function can be used for advanced iteration

```
import numpy as np
A = np.array([[1, 2, 3],
              [4, 5, 6]])
for x in A:
    print(x)
for x in A.flat:
    print(x)
```

```
[1 2 3]
[4 5 6]
1
2
3
4
5
6
```

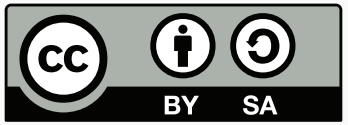
Types in NumPy

- python is rather limited in terms of numerical types
- scientific applications need precise control
- NumPy provides a large collection of types and automatic ways of handling them
- types are described by `dtype` objects complemented by a set of fundamental types of scalars
- when an array is created, a `dtype` parameter can be used to specified the type of the content
- default types
  - `float64` for decimals
  - `int64` for integers
  - `bool` for logical values

Numerous other concepts

- NumPy is a very rich library
- missing aspects
  - a lot!
  - proper use of types
  - dozens functions and methods
  - interactions between ufunc and other concepts (e.g. broadcasting and types)
  - memory layout
  - advanced aspects of iteration, indexing, etc.

Licence



Version

Last git commit: 2020-02-19
By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)
Git hash: 9e2d57088730bd3a349bda83bc9f96dde98391f8

Changelog

- February 2020: added NumPy
- December 2019:
  - added dictionaries
  - added tuples
  - added sets
- November 2019: initial version