Beyond arrays: other ways of collecting data in Python

Scientific Computing 2, AIMS, 2013
day_04

With `arrays`:

We are now used to having some of the following array properties:

With `arrays`:

We are now used to having some of the following array properties:

1. Known length and shape/dimension

With `arrays`:

We are now used to having some of the following array properties:

1. Known length and shape/dimension
2. Ordered values (use index to access)

With `arrays`:

We are now used to having some of the following array properties:

1. Known length and shape/dimension
2. Ordered values (use index to access)
3. Ability to update/change element values at any point during a program (reassigning values)

With `arrays`:

We are now used to having some of the following array properties:

1. Known length and shape/dimension
2. Ordered values (use index to access)
3. Ability to update/change element values at any point during a program (reassigning values)
4. Stores (lots of) a single data type; for the numerical arrays used, things like `int`, `float` and `complex`.

With `arrays`:

We are now used to having some of the following array properties:

1. Known length and shape/dimension
2. Ordered values (use index to access)
3. Ability to update/change element values at any point during a program (reassigning values)
4. Stores (lots of) a single data type; for the numerical arrays used, things like `int`, `float` and `complex`.

Buy maybe:

There will be times where other properties would be useful:

- storing non-numerical values such as letters, matching names+phone numbers, etc.;

Buy maybe:

There will be times where other properties would be useful:

- storing non-numerical values such as letters, matching names+phone numbers, etc.;
- being able to add more data values dynamically (while running code), to adjust to specific conditions;

Buy maybe:

There will be times where other properties would be useful:

- storing non-numerical values such as letters, matching names+phone numbers, etc.;
- being able to add more data values dynamically (while running code), to adjust to specific conditions;
- maybe even having constant collection values.

Python lists

Python lists 'relax' some of stringent rules of arrays, in particular properties #1 and 4, above.

Python lists

Python lists 'relax' some of stringent rules of arrays, in particular properties #1 and 4, above.
That is, one can change the length of lists and also use them to simultaneously store combinations of:
ints,
floats,
strings,
other lists,
etc.

Python lists

Python lists 'relax' some of stringent rules of arrays, in particular properties #1 and 4, above.

That is, one can change the length of lists and also use them to simultaneously store combinations of:

ints,

floats,

strings,

other lists,

etc.

Lists remain *ordered*, so that indexing works in the same way as for arrays.

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append('Example')
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append('Example')
>>> A.append(4)
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append('Example')
>>> A.append(4)
>>> A.append('day')
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append('Example')
>>> A.append(4)
>>> A.append('day')
>>> A.append(['with', 'some',' contents'])
>>> A
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
(4, int)
```

Python list operation examples

```
>>> A = []     # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
(4, int)
>>> A[3], type(A[3])
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
(4, int)
>>> A[3], type(A[3])
([’with’, ’some’, ’ contents’], list)
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
(4, int)
>>> A[3], type(A[3])
([’with’, ’some’, ’ contents’], list)
>>> A[3][1]
```

Python list operation examples

```
>>> A = []    # an empty list:  square brackets
>>> type(A)
list
>>> A.append(’Example’)
>>> A.append(4)
>>> A.append(’day’)
>>> A.append([’with’, ’some’,’ contents’])
>>> A
[’Example’, 4, ’day’, [’with’, ’some’, ’ contents’]]
>>> A[0], type(A[0])
(’Example’, str)
>>> A[1], type(A[1])
(4, int)
>>> A[3], type(A[3])
([’with’, ’some’, ’ contents’], list)
>>> A[3][1]
’some’
```

Python list operations

A subset from `help(list)`:

```
| append(...)
| L.append(object) -- append object to end
|
| count(...)
| L.count(value) -> integer -- return number of
occurrences of value
|
| extend(...)
| L.extend(iterable) -- extend list by appending elements
from the iterable
|
| index(...)
| L.index(value, [start, [stop]]) -> integer -- return
first index of value.
| Raises ValueError if the value is not present.
...
| sort(...)
| L.sort(cmp=None, key=None, reverse=False) -- stable sort
*IN PLACE*
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = ['list','one','of','stuff']    # type:  list
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = ['list','one','of','stuff']    # type:  list
>>> E = D
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = ['list','one','of','stuff']     # type:  list
>>> E = D
>>> E.insert(3, 'more')
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = ['list','one','of','stuff']     # type:  list
>>> E = D
>>> E.insert(3, 'more')
>>> E[1] = 'two'
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
>>> D
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
>>> D
[’list’, ’two’, ’of’, ’more’, ’stuff’]
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
>>> D
[’list’, ’two’, ’of’, ’more’, ’stuff’]
```

Make sense?

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
>>> D
[’list’, ’two’, ’of’, ’more’, ’stuff’]
```

Make sense?

If you want a *whole ’nother list* copied, then, e.g.:

```
>>> E = list(D)
```

Python list note

Lists are mutable, as arrays are, so you have to be careful when copying:

```
>>> D = [’list’,’one’,’of’,’stuff’]    # type:  list
>>> E = D
>>> E.insert(3, ’more’)
>>> E[1] = ’two’
>>> E
[’list’, ’two’, ’of’, ’more’, ’stuff’]
>>> D
[’list’, ’two’, ’of’, ’more’, ’stuff’]
```

Make sense?

If you want a *whole ’nother list* copied, then, e.g.:

```
>>> E = list(D)
```

(You can repeat the above with this and see what happens.)

Python strings

Another type in Python (which has actually been used above) is `string`.

Python strings

Another type in Python (which has actually been used above) is `string`.

These are defined using quotation marks:

```
>>> x=’hi!’
>>> y=’’hello!’’
>>> z=’’’how are ’ya!’’’
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=''hello!''
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=''hello!''
>>> y[3]
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=''hello!''
>>> y[3]
'l'
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=’’hello!’’
>>> y[3]
’l’
```

however, they are not *mutable* to change elements:

```
>>> y[3]=’L’
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=’’hello!’’
>>> y[3]
’l’
```

however, they are not *mutable* to change elements:

```
>>> y[3]=’L’
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
/Users/user/Desktop/TEACHING/paper2prog/AIMS_2013/making/<ipython-input-42-c985ef716b72> in
<module>()
----> 1 y[3]=’L’

TypeError:  ’str’ object does not support item assignment
```

Python string properties

Again, strings are *ordered* and can be accessed with indices:

```
>>> y=''hello!''
>>> y[3]
'l'
```

however, they are not *mutable* to change elements:

```
>>> y[3]='L'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
/Users/user/Desktop/TEACHING/paper2prog/AIMS_2013/making/<ipython-input-42-c985ef716b72> in
<module>()
----> 1 y[3]='L'

TypeError:  'str' object does not support item assignment
```

Things like slice selection with, e.g., `[:3]`, still work however.

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = ['I', 'am', 'not']    # some [list]
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = [’I’, ’am’, ’not’]    # some [list]
>>> A+[’alone’]    # append another list
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = ['I', 'am', 'not']     # some [list]
>>> A+['alone']     # append another list
['I', 'am', 'not', 'alone']
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = [’I’, ’am’, ’not’]    # some [list]
>>> A+[’alone’]    # append another list
[’I’, ’am’, ’not’, ’alone’]
>>> S = ’I have friends!  ’     # some ’str’
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = ['I', 'am', 'not']     # some [list]
>>> A+['alone']     # append another list
['I', 'am', 'not', 'alone']
>>> S = 'I have friends!  '      # some 'str'
>>> S*5 # repeat string
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = [’I’, ’am’, ’not’]    # some [list]
>>> A+[’alone’]    # append another list
[’I’, ’am’, ’not’, ’alone’]
>>> S = ’I have friends!  ’     # some ’str’
>>> S*5 # repeat string
’I have friends!  I have friends!  I have friends!  I
have friends!  I have friends!  ’
```

Python string/list properties

NB: a couple nice ways to append, which works for both strings and lists.

```
>>> A = [’I’, ’am’, ’not’]    # some [list]
>>> A+[’alone’]    # append another list
[’I’, ’am’, ’not’, ’alone’]
>>> S = ’I have friends!  ’     # some ’str’
>>> S*5 # repeat string
’I have friends!  I have friends!  I have friends!  I
have friends!  I have friends!  ’
```

Both the + and * have similar operation for lists and strings.

Conclusion

- So, lots to explore with lists and strings

Conclusion

- So, lots to explore with lists and strings

  ⇒ these come under the grouping of 'sequences', along with `tuples`

Conclusion

- So, lots to explore with lists and strings
  ⇒ these come under the grouping of 'sequences', along with `tuples`
- There are lots of operations to use with lists and strings, making them powerful computing tools...

Conclusion

- So, lots to explore with lists and strings
  ⇒ these come under the grouping of 'sequences', along with `tuples`
- There are lots of operations to use with lists and strings, making them powerful computing tools...
- But, with great power, comes great time/overhead sometimes computing. They might not always be the *fastest* things to use when working.

Conclusion

- So, lots to explore with lists and strings
  ⇒ these come under the grouping of 'sequences', along with `tuples`
- There are lots of operations to use with lists and strings, making them powerful computing tools...
- But, with great power, comes great time/overhead sometimes computing. They might not always be the *fastest* things to use when working.
- As with all computing, one must decide on trade-offs of ease, functionality, hardware, and sleep deprivation.