PYTHON LISTS, TUPLES and METHODS

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 9, 2023

Contents

1 Overview

- The `list` data type
- Working with lists
- Augmented assignment operators
- Methods as type-specific functions
- Standard Python vs. NumPy vs. pandas
- Magic 8 Ball reloaded

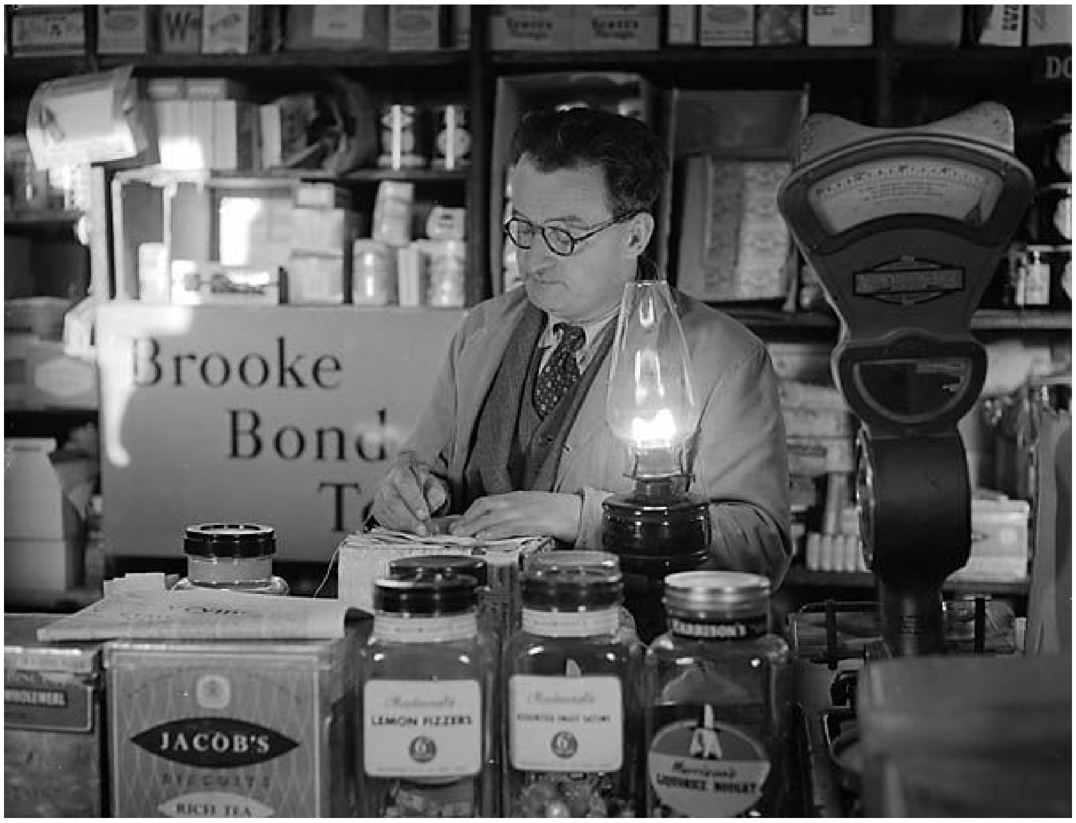


Figure 1: Llyfrgell Genedlaethol Cymru / Llanfachraeth in darkness (1957)

2 The `list` data type

- A `list` contains multiple values in an ordered sequence.
- A `list` is a *value* and can be stored in an object, and it also contains values also called *items*.
- The list items can be of any data type including lists:

```
print([1,2,3])   # numeric list (numeric items)
print(['cat','bat','rat','elephant'])    # string list (string items)
print(['hello', True, None, 42, 3.1415]) # mixed type list
```

```
[1, 2, 3]
['cat', 'bat', 'rat', 'elephant']
['hello', True, None, 42, 3.1415]
```

- Lists can be stored like any other value:

```
spam = ['cat', 'bat', 'rat']
print(len(spam))    # number of items in spam
print(type(spam))   # class of spam
print([] == list('')) # empty list
```

```
3
<class 'list'>
True
```

- `spam` is four things:
  1. a `list` variable (storage)
  2. a `list` value (stored)
  3. an ordered sequence of string values (indexed)
  4. an object (instanced)

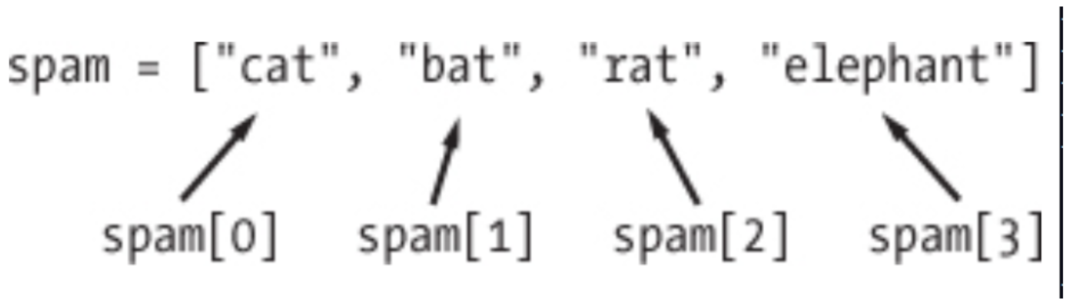


Figure 2: A list with its index values

3 Practice list creation, extraction and deletion

You should be able to do all of these exercises with what you learnt in the DataCamp course "Introduction to Python" (notebook in GitHub):

1. Assign these items to `spam` and extract them using a ranged `for` loop on one line separated by a single space: `cat bat rat elephant`

   ```
   spam = ['cat', 'bat', 'rat', 'elephant']
   for i in range(4):
       print(spam[i], end=' ')

   cat bat rat elephant
   ```

2. What if the list has `N` elements? Can you generalize the loop?

   ```
   spam = ['cat', 'bat', 'rat', 'elephant']
   for i in range(len(spam)):
       print(spam[i], end=' ')

   cat bat rat elephant
   ```

3. Use elements of `spam` to print the sentence `'The bat ate the cat.'` formatted with an f-string:

   ```
   spam = ['cat', 'bat', 'rat', 'elephant']
   print(f"'The {spam[1]} ate the {spam[0]}.'")

   'The bat ate the cat.'
   ```

4. Which error do you get when you use an index that exceeds the number of values in your list value? Create an example.

```
spam = ['cat', 'bat', 'rat', 'elephant']
print(spam[5])
```

5. Can index values be non-integer? Find out!

```
spam = ['cat', 'bat', 'rat', 'elephant']
print(spam[int(1.0)])
print(spam[1.0])
```

```
bat
```

6. How can you extract the last number in this list of lists?

```
spam = [['cat','bat'], [10,20,30,40,50]]
```

```
spam = [['cat','bat'], [10,20,30,40,50]]
print(spam[1][4],
      spam[1][-1],
      spam[-1][4],
      spam[-1][-1],
      end='')
```

```
50 50 50 50
```

7. Write `'The elephant is afraid of the bat.'` using *negative* indices of `spam = ['cat', 'bat', 'rat', 'elephant']`:

```
spam = ['cat', 'bat', 'rat', 'elephant']
print(f"'The {spam[-1]} is afraid of the {spam[-4]}.'")
```

```
'The elephant is afraid of the cat.'
```

8. From `spam = ['cat', 'bat', 'rat', 'elephant']`, extract `['cat','bat','rat']`:

```
spam = ['cat', 'bat', 'rat', 'elephant']
print(spam[0:3],    # slicing first three elements
      spam[-4:-1],  # slicing first three elements 'from the rear'
      sep='\n')
del spam[-1]        # deleting the last element
print(spam)
```

```
['cat', 'bat', 'rat']
['cat', 'bat', 'rat']
['cat', 'bat', 'rat']
```

9. Change `spam = ['cat', 'bat', 'rat', 'elephant']` to the list `['cat','armadillo','rat', 'armadillo']`:

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[-1] = 'armadillo'
print(spam)
spam[1] = 'armadillo'
print(spam)
```

```
['cat', 'bat', 'rat', 'armadillo']
['cat', 'armadillo', 'rat', 'armadillo']
```

10. Create `spam = ['cat', 'bat', 'cat', 'bat']` by list concatenation and replication:

```
spam = ['cat','bat'] * 2
print(spam)
del spam
spam = ['cat','bat'] + ['cat','bat']
print(spam)
```

```
['cat', 'bat', 'cat', 'bat']
['cat', 'bat', 'cat', 'bat']
```

4 Working with lists - `allMyCats`

- Here is a `list`-less version of a program to get the names of six cats from the user and printing them (pythontutor):

```
catName1 = input('Enter the name of cat 1: ')
catName2 = input('Enter the name of cat 2: ')
catName3 = input('Enter the name of cat 3: ')
catName4 = input('Enter the name of cat 4: ')
catName5 = input('Enter the name of cat 5: ')
catName6 = input('Enter the name of cat 6: ')
print(f'The cat names are: {catName1}, {catName2},\
 {catName3}, {catName4}, {catName5}, {catName6}')
```

```
Enter the name of cat 1:
```

- Instead, use a single variable that contains a `list` value (pythontutor):

```
catNames = []
while True:
    print('Enter name of cat (or nothing to stop):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]
if not catNames:
    print('You should get a cat')
else:
    print('The cat names are:')
    for name in catNames:
        print(f'{name}')
```

```
Enter name of cat (or nothing to stop):
```

1. Initialize empty list `catNames`
2. Infinite loop: ask for cat's `name` until empty entry
3. Check if `catNames` were entered
4. If `catNames` were entered, print them looping over the `list`

5 Looping over lists

- Notice how the `for` loop ranges over the list elements without `range`:

```
for i in ['a','b', None, 10,100]:
    print(i,end=' ')
```

```
a b None 10 100
```

- Can you print this list using a `for` loop with `range`?

```
List = ['a','b', None, 10,100]
for i in range(len(List)):
    print(List[i],end=' ')
```

```
a b None 10 100
```

- Instead of using `range` to get the integer index of the list items, call `enumerate` instead:

```
List = ['a','b', None, 10,100]
for index, item in enumerate(List):
    print(f'Index {index} in the list is: {item}')
```

```
Index 0 in the list is: a
Index 1 in the list is: b
Index 2 in the list is: None
Index 3 in the list is: 10
Index 4 in the list is: 100
```

- There is no simple way to get the name of `List` once it's been created because the variable name is just a *reference* to the data.
- All `global` objects are available in a *dictionary* `globals().items()`.

```
print(globals().items())
```

```
dict_items([('__name__', '__main__'), ('__doc__', None), ('__package__', None), (
```

6 Scope and lists

- Challenge:

  1. copy the code cell into a new code cell in Colab
  2. wrap the input routine into a function `getCatNames()`
  3. make `catNames` global
  4. call `getCatNames` before the final printout.

```
def getCatNames():
    global catNames  # make ‘catNames‘ global
    catNames = [ ]
    while True:
        print('Enter name of cat (or nothing to stop):')
        name = input()
        if name == '':
            return
        catNames = catNames + [name]
    return catNames

getCatNames()   # function call

if not catNames:
    print('You should get a cat')
else:
    print('The cat names are:')
    for name in catNames:
        print(f'{name}')
```

```
Enter name of cat (or nothing to stop):
```

- How could you keep `catNames` in local scope (inside the function) and still access its values outside?

```
def getCatNames():
    catNames = [ ]
    while True:
        print('Enter name of cat (or nothing to stop):')
        name = input()
        if name == '':
            return catNames
        catNames = catNames + [name]

myCatNames = getCatNames()
print(myCatNames)
```

```
Enter name of cat (or nothing to stop):
```

1. This function returns from the loop (and from the function call) when an empty string is entered (no input).
2. Otherwise it keeps adding cat names to the `catNames` list.
3. Upon returning from the function call, the list `catNames` is destroyed, but when the function call is saved in an object `myCatNames`, this object will hold the `return` value from `getCatNames`.

7 `in` or `out`?

- The `in` or `not in` command works on lists:

```
spam = ['cat', 'bat', 'rat']
print('cat' in spam)
print('chicken' not in spam)
```

```
True
True
```

8 Practice the `in` keyword for lists

- Write a script that lets the user type in a pet `name` and checks if the `name` is `in` a list `myPets` (which you need to create first). If it is `in` the list, say "I have a pet with that name", otherwise say that you don't.

- Solution:

```
myPets = ['Nanny', 'Rosie', 'Poppy', 'Jack']
name = input('Enter a pet name: ')
if name not in myPets:
    print(f"I don’t have a pet named {name}.")
else:
    print(f"{name} is my pet.")
```

```
Enter a pet name:
```

- Here I put the `input` command in a function `getPetName`. When it is called, it returns `name`, but `name` is local to the function, and you need to transfer it to the global variable `petName` to be used:

```
def getPetName():
    name = input('Enter a pet name: ')
    return name

myPets = ['Nanny', 'Rosie', 'Poppy', 'Jack']

petName = getPetName()

if petName not in myPets:
    print(f"I don’t have a pet named {petName}.")
else:
    print(f"{petName} is my pet.")
```

```
Enter a pet name:
```

9 Multiple assignments (`tuple` unpacking)

- You can assign multiple variables with the values in one line.
- The one assignment per line way:

```
cat = ['fast', 'moody', 'black']
speed = cat[0]
disposition = cat[1]
color = cat[2]
print(f'The {color} cat is {speed} and {disposition}')
```

```
The black cat is fast and moody
```

- Multiple assignments: number of variables and length of list must be exactly equal otherwise you get a `ValueError`.

```
cat = ['fast', 'moody', 'black']
speed, disposition, color = cat # stored as tuple and unpacked
print(f'The {color} cat is {speed} and {disposition}')

The black cat is fast and moody
```

- Handle the `ValueError` that is caused by adding a variable `name` to the assignment:

```
cat = ['fast', 'moody', 'black']
speed, disposition, color, name = cat # name is not known
print(f'The {color} cat is {speed} and {disposition}')
```

- Solution:

  1. put the assignment into a `try` clause and add a `except ValueError:` clause
  2. to test, run original version (exception), then add `'Jack'` to `cat` in the first line

```
cat = ['fast', 'moody', 'black']
try:
    speed, disposition, color, name = cat
except ValueError:
    print('ValueError - check multiple assignment')
else:
    print(f'The {color} cat named {name} is {speed} and {disposition}')

ValueError - check multiple assignment
```

10 Lists as `random` arguments

- The `random.choice` function will return a randomly selected item from the list:

```
import random
pets = ['dog', 'cat', 'squirrel','moose','mouse','pony','snake']
print(random.choice(pets))
```

```
squirrel
```

- This is a shorter form of `pets[random.randint(0,len(pets)-1]`:

  ```
  import random
  pets = ['dog', 'cat', 'squirrel','moose','mouse','pony','snake']
  print(pets[random.randint(0,len(pets)-1)])
  ```

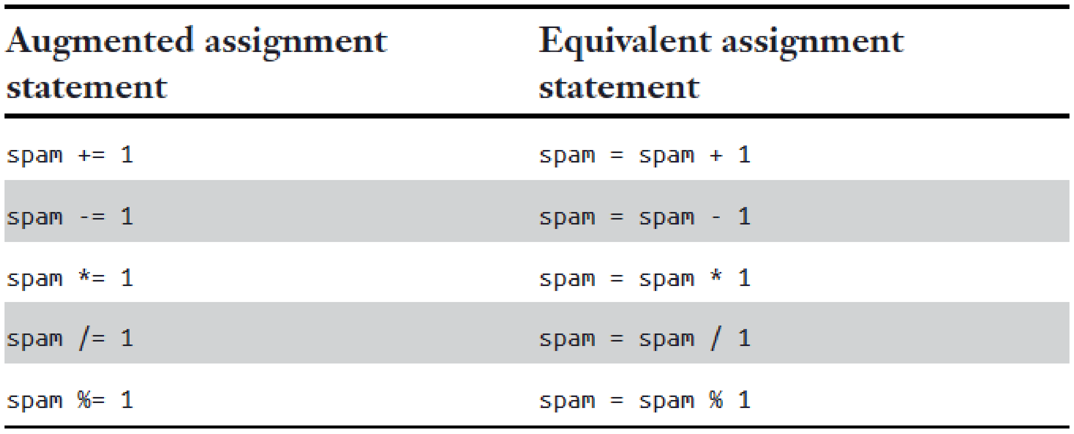
  ```
  dog
  ```

- The `random.shuffle` function will reorder the items in a list: it modifies the list *in place* rather than returning a new list.

  ```
  import random
  people = ['Alice', 'Bob', 'Carol', 'David']
  random.shuffle(people)
  print(people)
  ```

  ```
  ['Alice', 'Carol', 'Bob', 'David']
  ```

11 Augmented assignment operators



Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

Figure 3: Augmented assignment operators

- These operators work for numbers, strings and lists:

```
spam = 'Hello, '
spam += 'world!'   # equivalent to spam = spam + 'world!'
print(spam)

bacon = ['Huzza']
bacon *= 3         # equivalent to bacon = bacon * 3
print(bacon)
```

```
Hello, world!
['Huzza', 'Huzza', 'Huzza']
```

12 Methods for specific data types

- Each data type as its own set of methods, e.g. the `list` data type has methods for finding, adding, removing and manipulating values.
- Examples:
  1. to call the `list` method `index` on the item `'hello'` of a list `spam`:

     ```
     spam = ['hello','world']
     print(spam.index('hello'))  # returns an index
     ```

     ```
     0
     ```

  2. to call the `str` method `count` on the substring `'_'` of the string `'hello_world'` stored in `ham`:

     ```
     ham = 'hello_world'
     print(ham.count('_'))  # returns a count
     ```

     ```
     1
     ```

- This approach transfers to other packages such a `numpy` or `pandas` - the focus of the methods is on the library purpose like numeric data processing or statistical analysis.
- Where applicable, I will contrast standard Python with NumPy and/or pandas (Kudos OpenAI: ChatGPT has been invaluable for this task.)

13 Finding a value in a `list` with `index`

- If the value is not in the list, a `ValueError` is raised:

```
spam = ['hello', 'hi', 'howdy', 'hey']
print(spam.index('howdy'))
print(spam.index('howdy howdy howdy'))
```

```
2
```

- When there are duplicates, the first instance is returned:

```
spam = ['hello', 'hi', 'howdy', 'hey', 'hi']
print(spam.index('hi'))
```

```
1
```

14 Finding a value in a numpy `array` with `where`

- In NumPy, you can use the `where` function - a lot more information is available, but you need more skill to sort through it:

```
import numpy as np
spam = ['hello', 'hi', 'howdy', 'hey', 'hi']

# turn list into numpy array
spam_np = np.array(spam)

# store value of index for item
idx = np.where(spam_np == 'howdy')

print(idx)    # index informaion (full)
print(idx[0][0])  # index only
print(spam_np[idx])   # array value
```

```
(array([2], dtype=int64),)
2
['howdy']
```

15 Finding a value in a pandas `series` with `pd.index`

- In pandas, you can use Boolean indexing:

```
import pandas as pd

# Create a pandas Series
spam_pd = pd.Series(['hello', 'hi', 'howdy', 'hey', 'hi'])

# Find the index where the value is equal to 'howdy'
index = spam_pd[spam_pd == 'howdy'].index[0]

print(index)
```

```
2
```

- If the value is not found in the Series, it will raise an `IndexError`.

16 Adding values for lists with `append` and `insert`

- You can add new values to a list with `append` (at the end) and `insert`.
- Append `'moose'` at the end of `spam`:

```
spam = ['cat', 'dog', 'bat']
print(spam)
spam.append('moose')
print(spam)
```

```
['cat', 'dog', 'bat']
['cat', 'dog', 'bat', 'moose']
```

- Insert `'chicken'` as item number `1` into `spam`:

```
spam = ['cat', 'dog', 'bat']
print(spam)
spam.insert(1,'chicken')
print(spam)
```

```
['cat', 'dog', 'bat']
['cat', 'chicken', 'dog', 'bat']
```

- These functions modify a list *in place*: neither of them gives the new value as a return value - they return `None` instead:

```
spam = ['cat', 'dog', 'bat']
print(spam.append('moose'))
print(spam)
print(spam.insert(1,'chicken'))
print(spam)
```

```
None
['cat', 'dog', 'bat', 'moose']
None
['cat', 'chicken', 'dog', 'bat', 'moose']
```

- If that's so, what does `spam = spam.append('elephant')` do?

```
spam = ['cat', 'dog', 'bat']
print(spam)
spam = spam.append('elephant')
print(spam)
```

```
['cat', 'dog', 'bat']
None
```

17 Adding and inserting for NumPy `array`

- By contrast, NumPy's `np.append` and `np.insert` methods create a new array and you need to assign the result back to the array to keep it:

```
import numpy as np

spam_np = np.array(['cat', 'dog', 'bat', 'elephant'])

print(spam_np)
```

```
spam_np = np.append(spam_np, 'moose')

print(spam_np)

spam_np = np.insert(spam_np, 1, 'chicken')

print(spam_np)
```

```
['cat' 'dog' 'bat' 'elephant']
['cat' 'dog' 'bat' 'elephant' 'moose']
['cat' 'chicken' 'dog' 'bat' 'elephant' 'moose']
```

- The behavior of NumPy for strings is tricky though: e.g. string items in the array will be truncated if the inserted string is larger than the largest string already in the array.
- To test that, run the code above and remove `'elephant'`: the resulting inserted array will list `'chick'` and not `'chicken'`.
- Numbers work better: an example with `np.append`

```
import numpy as np

# Create a numpy array
arr = np.array([1, 2, 3, 4, 5])

# Append a single value
arr = np.append(arr, 6)
print(arr)  # Output: [1 2 3 4 5 6]

# Append multiple values
arr = np.append(arr, [7, 8, 9])
print(arr)  # Output: [1 2 3 4 5 6 7 8 9]
```

```
[1 2 3 4 5 6]
[1 2 3 4 5 6 7 8 9]
```

- An example with `np.insert`:

```
import numpy as np

# Create a numpy array
arr = np.array([1, 2, 3, 4, 5])

# Insert a single value at index 2
arr = np.insert(arr, 2, 6)
print(arr)  # Output: [1 2 6 3 4 5]

# Insert multiple values at index 3
arr = np.insert(arr, 3, [7, 8, 9])
print(arr)  # Output: [1 2 6 7 8 9 3 4 5]
```

```
[1 2 6 3 4 5]
[1 2 6 7 8 9 3 4 5]
```

18 Adding columns and rows in pandas `DataFrame`

- The central structure for `pandas` is the DataFrame, a tabular structure of column vectors of the same length with each vector only having one type.
- Let's import `pandas` as `pd` and create a DataFrame `df`:

```
import pandas as pd

# Create a DataFrame of four column vectors A,B,C,D
df = pd.DataFrame({
    'A': ['foo', 'bar', 'baz'],
    'B': ['one', 'one', 'two'],
    'C': ['x', 'y', 'z'],
    'D': [1, 2, 3]
})
```

- Adding a new column to a DataFrame by adding it like an index:

```
import pandas as pd

# Create a DataFrame of four column vectors A,B,C,D
df = pd.DataFrame({
    'A': ['foo', 'bar', 'baz'],
    'B': ['one', 'one', 'two'],
    'C': ['x', 'y', 'z'],
    'D': [1, 2, 3]
})
# Add a new column E
df['E'] = ['alpha', 'beta', 'gamma']

print(df)

     A    B  C  D      E
0  foo  one  x  1  alpha
1  bar  one  y  2   beta
2  baz  two  z  3  gamma
```

- Inserting a new column at a specific position with `df.insert`:

```
import pandas as pd

# Create a DataFrame of four column vectors A,B,C,D
df = pd.DataFrame({
    'A': ['foo', 'bar', 'baz'],
    'B': ['one', 'one', 'two'],
    'C': ['x', 'y', 'z'],
    'D': [1, 2, 3]
})
# Insert a new column at position 1
df.insert(1, 'F', ['apple', 'banana', 'cherry'])

print(df)

     A       F    B  C  D
0  foo   apple  one  x  1
1  bar  banana  one  y  2
2  baz  cherry  two  z  3
```

- Adding a new row to a DataFrame with `df.concat` (`df.append` is also available but it is deprecated as of 2022):

```
import pandas as pd

# Create a DataFrame of four column vectors A,B,C,D
df = pd.DataFrame({
    'A': ['foo', 'bar', 'baz'],
    'B': ['one', 'one', 'two'],
    'C': ['x', 'y', 'z'],
    'D': [1, 2, 3]
})
# Create a new DataFrame for the new row
new_row = pd.DataFrame([{'A': 'qux', 'B': 'three', 'C': 'w',\
                         'D': 4, 'E': 'delta', 'F': 'durian'}])

# Use pd.concat to append the new row
df = pd.concat([df, new_row])

# Use pd.append to append the new row once again
df = df.append(new_row)

print(df)

     A      B  C  D      E       F
0  foo    one  x  1    NaN     NaN
1  bar    one  y  2    NaN     NaN
2  baz    two  z  3    NaN     NaN
0  qux  three  w  4  delta  durian
0  qux  three  w  4  delta  durian
```

19 Trying to call a method on another data type

- The `append` and `insert` methods are `list` methods and won't work for strings or integers:

```
eggs = 'hello'
eggs.append('world')
```

- Calling `insert` on an integer:

```
bacon = 42
bacon.insert(1,'world')
```

20 Removing values from lists with `remove` or `del`

- The `remove` method removes its arguments in place:

  ```
  spam = ['cat','bat','rat','elephant']
  print(spam)
  spam.remove('bat')
  print(spam)

  ['cat', 'bat', 'rat', 'elephant']
  ['cat', 'rat', 'elephant']
  ```

- Trying to remove a value that does not exist raises a `ValueError`:

  ```
  spam = ['cat','bat','rat','elephant']
  spam.remove('chicken')
  ```

- If there are multiple identical items, only the first will be removed:

  ```
  spam = ['cat','bat','rat','elephant','cat','bat']
  print(spam)
  spam.remove('bat')
  print(spam)   # only the first instance is removed

  ['cat', 'bat', 'rat', 'elephant', 'cat', 'bat']
  ['cat', 'rat', 'elephant', 'cat', 'bat']
  ```

- Wondering at this point how many values you can remove at a time? Check the help (don't forget that this is a `list` method):

  ```
  help(list.remove)

  Help on method_descriptor:

  remove(self, value, /)
      Remove first occurrence of value.

      Raises ValueError if the value is not present.
  ```

- If you know the index of the item you want to remove, you can use the `del` keyword to delete items:

```
spam = ['cat','bat','rat','elephant','cat','bat']
del spam[1]
print(spam)

['cat', 'rat', 'elephant', 'cat', 'bat']
```

- To remove more than one item at a time, you can either use a `list` comprehension (`set` building), or the `filter` function (lambda):

```
spam = ['cat','bat','rat','elephant','cat','bat']

# Remove all 'bat' items
spam = [item for item in spam if item != 'bat']

print(spam)  # Output: ['cat', 'rat', 'elephant', 'cat']

['cat', 'rat', 'elephant', 'cat']
```

- In the example, the `filter` function (*iterator*) takes an anonymous or `lambda` function as the argument:

```
spam = ['cat','bat','rat','elephant','cat','bat']

# Remove all 'bat' items
spam = list(filter(lambda item: item != 'bat', spam))

print(spam)  # Output: ['cat', 'rat', 'elephant', 'cat']

['cat', 'rat', 'elephant', 'cat']
```

21 Removing values from a NumPy `array`

- You cannot directly remove an item from an `array` like in a Python list with `remove` but you can create a new array that doesn't include the items to be removed.

- Using Boolean indexing or masking:

```
import numpy as np

# Create a numpy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Create a new array that doesn't include the value 5
arr = arr[arr != 5]

print(arr)  # Output: [1 2 3 4 6 7 8 9]
```

```
[1 2 3 4 6 7 8 9]
```

- Using the `np.delete` method:

```
import numpy as np

# Create a numpy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Create a new array that doesn't include the item at index 4
arr = np.delete(arr, 4)

print(arr)  # Output: [1 2 3 4 6 7 8 9]
```

```
[1 2 3 4 6 7 8 9]
```

22 Removing values from a pandas `DataFrame`

- The `pd.drop` function is used to remove either columns or rows from a DataFrame: the keyword parameter `axis` is `1` for columns, `0` for rows.
- Unlike the NumPy arrays, you can specify if you wish to modify the DataFrame in place using the `inplace` keyword parameter.
- Remove a column:

```
import pandas as pd
```

```
# Create a simple dataframe
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

print("Original DataFrame")
print(df)

# Drop column 'A'
df = df.drop('A', axis=1)

print("DataFrame After Dropping Column 'A'")
print(df)
```

```
Original DataFrame
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
DataFrame After Dropping Column 'A'
   B  C
0  4  7
1  5  8
2  6  9
```

- Remove a row:

```
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

print("Original DataFrame")
print(df)
```

```
# Drop row at index 1
df = df.drop(1, axis=0)

print("DataFrame After Dropping Row at Index 1")
print(df)
```

```
Original DataFrame
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
DataFrame After Dropping Row at Index 1
   A  B  C
0  1  4  7
2  3  6  9
```

23 Sorting values in a list with `sort`

- Lists of number values or strings can be sorted with `list.sort`:

```
spam = [2, 5, 3.14, 1, -7]
spam.sort()  # default sort is ascending
print(spam)

ham = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
ham.sort()  # default sort is ascending in alphabetical order
print(ham)
```

```
[-7, 1, 2, 3.14, 5]
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

- To reverse the order from ascending to descending use the `reverse` keyword:

```
spam = [2, 5, 3.14, 1, -7]
spam.sort(reverse=True)  # reverse sorting
print(spam)
```

```
ham = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
ham.sort(reverse=True)  # reverse sorting
print(ham)
```

```
[5, 3.14, 2, 1, -7]
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

- As you can see in the `help(list.sort)` docstring, you can also sort using a function, e.g. the `len` function:

```
ham = ['ants', 'cats', 'dogs', 'badgers', 'elephants', 'snakes']
ham.sort(key=len,reverse=True)  # reverse sorting by length
print(ham)
```

```
['elephants', 'badgers', 'snakes', 'ants', 'cats', 'dogs']
```

- In the last example, note that `'ants'` goes before `'cats'` before `'dogs'` because within a group of strings with the same `len` value, sorting is alphabetical (in ascending order).
- To change this is more complex: you use an anonymous `lambda` function in the `sort` function that sorts first by `len` and then reverses the order:

```
ham = ['ants', 'cats', 'dogs', 'badgers', 'elephants', 'snakes']

# Sort the list by the number of characters in each string, and then reverse the a
ham.sort(key=lambda x: (len(x), x), reverse=True)

print(ham)
```

```
['elephants', 'badgers', 'snakes', 'dogs', 'cats', 'ants']
```

24 Reversing values in a list with `reverse`

- To quickly reverse the order of list items, use `list.reverse`:

```
spam = ['cat', 'dog', 'moose']
spam.reverse()
print(spam)
```

```
['moose', 'dog', 'cat']
```

- This is a simple function that does not offer any keyword parameters:

```
help(list.reverse)
```

```
Help on method_descriptor:

reverse(self, /)
    Reverse *IN PLACE*.
```

25 Sorting a NumPy array

- The `np.sort` function offers different sorting algorithms (`kind`), and you can specify along which dimension to sort (`axis`), and the `order`.

```
import numpy as np
help(np.sort)
```

```
Help on function sort in module numpy:

sort(a, axis=-1, kind=None, order=None)
    Return a sorted copy of an array.

    Parameters
    ----------
    a : array_like
Array to be sorted.
    axis : int or None, optional
Axis along which to sort. If None, the array is flattened before
sorting. The default is -1, which sorts along the last axis.
    kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
and 'mergesort' use timsort or radix sort under the covers and, in general,
the actual implementation will vary with data type. The 'mergesort' option
is retained for backwards compatibility.

.. versionchanged:: 1.15.0.
```

```
   The 'stable' option was added.

    order : str or list of str, optional
When `a` is an array with fields defined, this argument specifies
which fields to compare first, second, etc.  A single field can
be specified as a string, and not all fields need be specified,
but unspecified fields will still be used, in the order in which
they come up in the dtype, to break ties.

    Returns
    -------
    sorted_array : ndarray
Array of the same type and shape as `a`.

    See Also
    --------
    ndarray.sort : Method to sort an array in-place.
    argsort : Indirect sort.
    lexsort : Indirect stable sort on multiple keys.
    searchsorted : Find elements in a sorted array.
    partition : Partial sort.

    Notes
    -----
    The various sorting algorithms are characterized by their average speed,
    worst case performance, work space size, and whether they are stable. A
    stable sort keeps items with the same key in the same relative
    order. The four algorithms implemented in NumPy have the following
    properties:

    =========== ======= ============= ============ ========
       kind      speed   worst case    work space   stable
    =========== ======= ============= ============ ========
    'quicksort'    1     O(n^2)            0          no
    'heapsort'     3     O(n*log(n))       0          no
    'mergesort'    2     O(n*log(n))      ~n/2        yes
    'timsort'      2     O(n*log(n))      ~n/2        yes
    =========== ======= ============= ============ ========

    .. note:: The datatype determines which of 'mergesort' or 'timsort'
   is actually used, even if 'mergesort' is specified. User selection
   at a finer scale is not currently available.

All the sort algorithms make temporary copies of the data when
sorting along any but the last axis.  Consequently, sorting along
the last axis is faster and uses less space than sorting along
any other axis.

The sort order for complex numbers is lexicographic. If both the real
and imaginary parts are non-nan then the order is determined by the
real parts except when they are equal, in which case the order is
determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan
values led to undefined behaviour. In numpy versions >= 1.4.0 nan
values are sorted to the end. The extended sort order is:

  * Real: [R, nan]
  * Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]

where R is a non-nan real value. Complex values with the same nan
placements are sorted according to the non-nan part if it exists.
Non-nan values are sorted as before.

.. versionadded:: 1.12.0

quicksort has been changed to `introsort <https://en.wikipedia.org/wiki/Intros
When sorting does not make enough progress it switches to
`heapsort <https://en.wikipedia.org/wiki/Heapsort>`_.
This implementation makes quicksort O(n*log(n)) in the worst case.

'stable' automatically chooses the best stable sorting algorithm
for the data type being sorted.
It, along with 'mergesort' is currently mapped to
`timsort <https://en.wikipedia.org/wiki/Timsort>`_
or `radix sort <https://en.wikipedia.org/wiki/Radix_sort>`_
depending on the data type.
API forward compatibility currently limits the
ability to select the implementation and it is hardwired for the different
data types.
```

```
 .. versionadded:: 1.17.0

 Timsort is added for better performance on already or nearly
 sorted data. On random data timsort is almost identical to
 mergesort. It is now used for stable sort while quicksort is still the
 default sort if none is chosen. For timsort details, refer to
 `CPython listsort.txt <https://github.com/python/cpython/blob/3.7/Objects/list
 'mergesort' and 'stable' are mapped to radix sort for integer data types. Radi
 O(n) sort instead of O(n log n).

 .. versionchanged:: 1.18.0

 NaT now sorts to the end of arrays for consistency with NaN.

 Examples
 --------
 >>> a = np.array([[1,4],[3,1]])
 >>> np.sort(a)                # sort along the last axis
 array([[1, 4],
[1, 3]])
 >>> np.sort(a, axis=None)     # sort the flattened array
 array([1, 1, 3, 4])
 >>> np.sort(a, axis=0)        # sort along the first axis
 array([[1, 1],
[3, 4]])

 Use the `order` keyword to specify a field to use when sorting a
 structured array:

 >>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
 >>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
 ...           ('Galahad', 1.7, 38)]
 >>> a = np.array(values, dtype=dtype)       # create a structured array
 >>> np.sort(a, order='height')                        # doctest: +SKIP
 array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
('Lancelot', 1.8999999999999999, 38)],
dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])

 Sort by age, then height if ages are equal:
  >>> np.sort(a, order=['age', 'height'])               # doctest: +SKIP
  array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
 ('Arthur', 1.8, 41)],
dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

- A simple example - sorting is in ascending order by default, and a new sorted array is created.

```
import numpy as np

# Create a numpy array
arr = np.array([3, 2, 1, 5, 4])

print("Original array:")
print(arr)

# Sort the array
sorted_arr = np.sort(arr)

print("Sorted array:")
print(sorted_arr)
```

```
Original array:
[3 2 1 5 4]
Sorted array:
[1 2 3 4 5]
```

- To reverse the sorting order of a NumPy array, you can use the `[::-1]` slicing operation after sorting the array:

```
import numpy as np

# Create a numpy array
arr = np.array([3, 2, 1, 5, 4])

print("Original array:")
print(arr)
```

```
# Sort the array in descending order
sorted_arr_desc = np.sort(arr)[::-1]

print("Array sorted in descending order:")
print(sorted_arr_desc)
```

```
Original array:
[3 2 1 5 4]
Array sorted in descending order:
[5 4 3 2 1]
```

- This slicing trick also works with lists:

```
spam = [3, 2, 1, 5, 4]
spam.sort()
print(spam)
print(spam[::-1])
```

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

26 Sorting a pandas `DataFrame`

- You can sort a DataFrame by values in one or more columns with the `pd.sort_values` method:

```
import pandas as pd

# create a simple dataframe with columns, A,B,C
df = pd.DataFrame({
    'A': [2,3,1],
    'B': [1,3,2],
    'C': ['b','a','c']
})

print("Original DataFrame")
print(df)
```

```
# Sort by column 'A'
df_sorted = df.sort_values('A')

print("DataFrame sorted by column 'A'")
print(df_sorted)

Original DataFrame
   A  B  C
0  2  1  b
1  3  3  a
2  1  2  c
DataFrame sorted by column 'A'
   A  B  C
2  1  2  c
0  2  1  b
1  3  3  a
```

- A DataFrame is not a matrix: to sort by the rows you need to sort by the row labels (the index) using the `sort_index` method:

```
import pandas as pd

# create a simple dataframe with columns, A,B,C
df = pd.DataFrame({
    'A': [2,3,1],
    'B': [1,3,2],
    'C': ['b','a','c']
}, index = ['Y', 'X', 'Z'])

print("Original DataFrame")
print(df)

# sort by index
df_sorted = df.sort_index()

print("DataFrame sorted by index")
print(df_sorted)

Original DataFrame
```

```
   A  B  C
Y  2  1  b
X  3  3  a
Z  1  2  c
DataFrame sorted by index
   A  B  C
X  3  3  a
Y  2  1  b
Z  1  2  c
```

27 Exceptions to Python indentation rules for `list`

- Indentation is significant in Python because the indentation for a line of code tells Python what block it is in, otherwise you get an `IndentationError`.
- Lists, however, can span several lines in any indentation, and the same goes for pandas `DataFrame` and NumPy `array` structures: Python knows that the structure isn't finished before the ending bracket.
- List example:

```
spam = ['apples',
        'oranges',
                        'bananas',
 'peaches'                            ]
print(spam)
print(type(spam))
```

```
['apples', 'oranges', 'bananas', 'peaches']
<class 'list'>
```

- NumPy example:

```
import numpy as np
arr = np.array(['apples',
        'oranges',
                        'bananas',
 'peaches'                            ])
print(arr)
print(type(arr))
```

```
['apples' 'oranges' 'bananas' 'peaches']
<class 'numpy.ndarray'>
```

- Pandas example:

```
import pandas as pd
df = pd.DataFrame({ 'A': [1,2,
                          3],
    'B' :
                    [4, 5, 6],
                       'C':
[7,8,9]
                    })
print(df)
print(type(df))
```

```
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
<class 'pandas.core.frame.DataFrame'>
```

28 Practice `list` methods - Magic 8 Ball reloaded

1. Earlier, you created a Magic 8 ball program as a fortune teller:

```
import random

def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidely so'
    elif answerNumber == 3:
        return 'It is Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

r = random.randint(1,9)
fortune = getAnswer(r)
print(fortune)

Concentrate and ask again
```

2. Using lists, write a much more elegant version of the previous Magic 8 Ball program:

   - instead of several lines of nearly identical `elif` statements, create a single list `messages` to work with. The list holds the messages as its items.
   - instead of calling a function `getAnswer`, `print` a message using `random.randint` to pick the index (i.e. the position) of the message - there are 9 messages. Remember that `random.randint(a,b)` picks an integer in `[a,b]`.
   - You can generalize the program further by making the upper bound of `random.randint` independent of the number 9. Now you could add messages to the list ad infinitum.

3. Solution:

```
import random

messages = ['It is certain',
            'It is decidedly so',
            'Yes, definitely',
            'Reply hazy try again',
            'Ask again later',
            'Concentrate and ask again',
            'My reply is no',
            'Outlook not so good',
            'Very doubtful']
print(messages[random.randint(0,len(messages)-1)])

My reply is no
```

4. Test the performance of both programs in Colab using `%timeit`. Do you record any difference?

29 Summary

- Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable.
- Lists are a sequence data type that is mutable, meaning that their contents can change.
- Tuples and strings, though also sequence data types, are immutable and cannot be changed.
- A variable that contains a tuple or string value can be overwritten with a new tuple or string value
- This is not the same thing as modifying the existing valuein place — like, say, the `append()` or `remove()` methods do on lists.
- Variables do not store list values directly; they store references to lists. Any change you make to a list may impact other variables.
- You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.
- NumPy array and pandas DataFrame structures are purpose-built to handle multi-dimensional numeric data (NumPy) or general data in tabular form (pandas).
- The methods to manipulate arrays and DataFrames in many ways parallel the functions for lists (often they have the same name).

30 Glossary

TERM/COMMAND	DEFINITION
`random.choice`	Return randomly selected list item
`random.shuffle`	Randomly shuffle list items
`np.array`	Numpy array creation
`list.append`	Append values to list *in place*
`list.insert`	Insert value at list index value *in place*
`np.append`	Create new array with appended value
`np.insert`	Create new array with inserted value
`df.insert`	Insert new column in pandas DataFrame
`df.concat`	Add new row to pandas DataFrame
`list.remove`	Remove values from list
`del`	Keyword to remove specific list value
Comprehension	Building Boolean index flags for sets
`lambda`	Keyword for anonymous functions
`filter`	Iterator to filter sequence data
`np.delete`	Create new array without the deleted value
`pd.drop`	Remove columns or values from DataFrame
`list.sort`	Sort list values in place (`reverse=False`)
`list.reverse`	Reverse list items in place
`np.sort`	Sort NumPy arrays
`[::-1]`	Reverse sorting order slicing (lists or arrays)
`pd.sort_values`	Sort DataFrame by values in one or more columns
`pd.sort_index`	Sort DataFrame by row index