MODULE 4
DICTIONARIES AND STRUCTURING DATA

1. The Dictionary Data Type

2. Pretty Printing

3. Using Data Structures to Model Real-World Things.

The Dictionary Data Type

- A dictionary is a collection of many values. Indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair. A dictionary is typed with braces, {}.
- A dictionary is typed with braces, {}.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

- This assigns a dictionary to the myCat variable. This dictionary's keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

- Dictionaries can still use integer values as keys, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

- Unlike lists, items in dictionaries are unordered.
- The first item in a list named spam would be spam[0]. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Trying to access a key that does not exist in a dictionary will result in a Key Error error message, much like a list's "out-of-range" IndexError error message.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

- We can have arbitrary values for the keys that allows us to organize our data in powerful ways.
- Ex: we want to store data about our friends' birthdays. We can use a dictionary with the names as keys and the birthdays as values.

Program:

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

Output:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

- We create an initial dictionary and store it in birthdays 1.
- We can see if the entered name exists as a key in the dictionary with the in keyword 2.
- If the name is in the dictionary, we access the associated value using square brackets 3; if not, we can add it using the same square bracket syntax combined with the assignment operator 4.

The keys(), values(), and items() Methods:

- There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items().
- Data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

- A for loop can iterate over the keys, values, or key-value pairs in a dictionary by using keys(), values(), and items() methods.
- The values in the dict_items value returned by the items() method are tuples of the key and value.

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

- If we want a true list from one of these methods, pass its list-like return value to the list() function.

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

- The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].
- We can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

We can use the in and not in operators to see whether a certain key or value exists in a dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

The get() Method

- Dictionaries have a get() method that takes two arguments:
- The key of the value to retrieve and
- A fallback value to return if that key does not exist.

The set default() Method

- To set a value in a dictionary for a certain key only if that key does not already have a value

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

- The setdefault() method offers a way to do this in one line of code.
- Setdeafault() takes 2 arguments:
    - The first argument is the key to check for, and
    - The second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value.

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

- The first time setdefault() is called, the dictionary in spam changes to {'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the value 'black' because this is now the value set for the key 'color'. When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white' because spam already has a key named 'color'.

Ex: program that counts the number of occurrences of each letter in a string.

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

- The program loops over each character in the message variable's string, counting how often each character appears.
- The setdefault() method call ensures that the key is in the count dictionary (with a default value of 0), so the program doesn't throw a KeyError error when count[character] = count[character] + 1 is executed.

Output:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

Pretty Printing

- Importing pprint module will provide access to the pprint() and pformat() functions that will "pretty print" a dictionary's values.
- This is helpful when we want a cleaner display of the items in a dictionary than what print() provides and also it is helpful when the dictionary itself contains nested lists or dictionaries..

Program: counts the number of occurrences of each letter in a string.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output:

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

Using Data Structures to Model Real-World Things

A Tic-Tac-Toe Board

- A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an X, an O, or a blank. To represent the board with a dictionary, we can assign each slot a string-value key as shown in below figure.

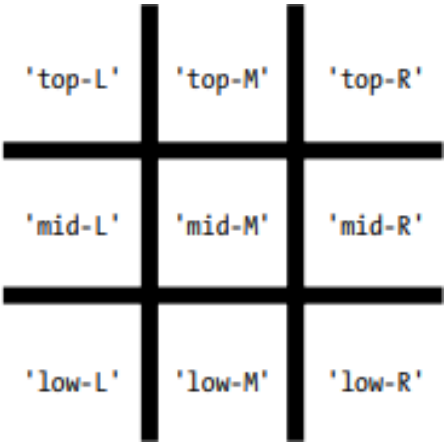


Figure: The slots of a tic-tactoe board with their corresponding keys

- We can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space character).
  - To store nine strings. We can use a dictionary of values for this.
  - The string value with the key 'top-R' can represent the top-right corner,
  - The string value with the key 'low-L' can represent the bottom-left corner,
  - The string value with the key 'mid-M' can represent the middle, and so on.

- Store this board-as-a-dictionary in a variable named theBoard.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

- The data structure stored in the theBoard variable represents the tic-tactoe board in the below Figure.

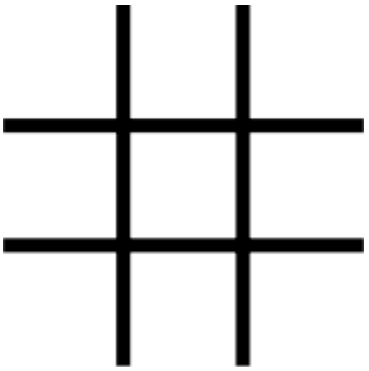


Figure: An empty tic-tac-toe board

- Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary as shown below:

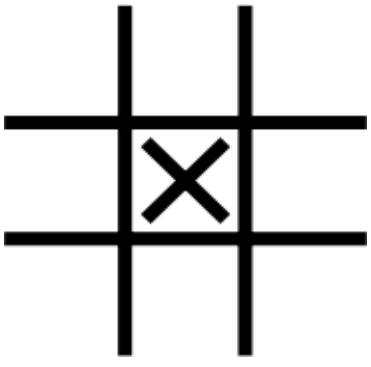


Figure: A first move

- A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

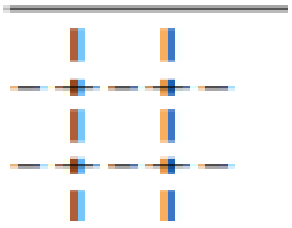
- The data structure in theBoard now represents tic-tac-toe board in the below Figure.

Figure: Player O wins.

- The player sees only what is printed to the screen, not the contents of variables.
- The tic-tac-toe program is updated as below.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:



- The printBoard() function can handle any tic-tac-toe data structure you pass it.

Program

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:

```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

- Now we created a data structure to represent a tic-tac-toe board and wrote code in printBoard() to interpret that data structure, we now have a program that "models" the tic-tac-toe board.
- **Program:** allows the players to enter their moves.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

Output:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

Nested Dictionaries and Lists

- We can have program that contains dictionaries and lists which in turn contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

Program: which contains nested dictionaries in order to see who is bringing what to a picnic

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

- Inside the totalBrought() function, the for loop iterates over the keyvalue pairs in guests 1.
- Inside the loop, the string of the guest's name is assigned to k, and the dictionary of picnic items they're bringing is assigned to v.
- If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to numBrought 2.
- If it does not exist as a key, the get() method returns 0 to be added to numBrought.

Output:

```
Number of things being brought:
 - Apples 7
 - Cups 3
 - Cakes 0
 - Ham Sandwiches 3
 - Apple Pies    1
```

Example 1: Sorting Dictionary By Key

```
myDict = {'ravi': 10, 'rajnish': 9,
          'sanjeev': 15, 'yash': 2, 'suraj': 32}

myKeys = list(myDict.keys())
myKeys.sort()
sorted_dict = {i: myDict[i] for i in myKeys}

print(sorted_dict)
```

output:

```
{'rajnish': 9, 'ravi': 10, 'sanjeev': 15, 'suraj': 32, 'yash': 2}
```

STRINGS

Working with strings :

String Literals

- String values begin and end with a single quote.
- But we want to use either double or single quotes within a string then we have multiple ways to do it as shown below.

Double Quotes

- One benefit of using double quotes is that the string can have a single quote character in it.

```
>>> spam = "That is Alice's cat."
```

- Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string.

Escape Characters

- If you need to use both single quotes and double quotes in the string, you'll need to use escape characters.
- An escape character consists of a backslash (\) followed by the character you want to add to the string.

```
>>> spam = 'Say hi to Bob\'s mother.'
```

- Python knows that the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" allows to put single quotes and double quotes inside your strings, respectively.

- **Ex:**

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

- The different special escape characters can be used in a program as listed below in a table.

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Raw Strings

- You can place an r before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Multiline Strings with Triple Quotes

- A multiline string in Python begins and ends with either three single quotes or three double quotes.
- Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string.

Program

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

Output

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

- The following print() call would print identical text but doesn't use a multiline string.

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line.

- A multiline string is often used for comments that span multiple lines

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

- Strings use indexes and slices the same way lists do. We can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

- The space and exclamation point are included in the character count, so 'Hello world!' is 12 characters long.
- If we specify an index, you'll get the character at that position in the string.

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

- If we specify a range from one index to another, the starting index is included and the ending index is not.

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

- The substring we get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the space at index 5.

Note: slicing a string does not modify the original string.

The in and not in Operators with Strings

- The **in** and **not in** operators can be used with strings just like with list values.
- An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

- These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

PUTTING STRINGS INSIDE OTHER STRINGS

Putting strings inside other strings is a common operation in programming. So far, we've been using the + operator and string concatenation to do this:

```
>>> name = 'Al'
>>> age = 4000
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
```

However, this requires a lot of tedious typing. A simpler approach is to use *string interpolation*, in which the %s operator inside the string acts as a marker to be replaced by values following the string. One benefit of string interpolation is that str() doesn't have to be called to convert values to strings. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' % (name, age)
'My name is Al. I am 4000 years old.'
```

Python 3.6 introduced *f-strings*, which is similar to string interpolation except that braces are used instead of %s, with the expressions placed directly inside the braces. Like raw strings, f-strings have an f prefix before the starting quotation mark. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> f'My name is {name}. Next year I will be {age + 1}.'
'My name is Al. Next year I will be 4001.'
```

Remember to include the f prefix; otherwise, the braces and their contents will be a part of the string value:

```
>>> 'My name is {name}. Next year I will be {age + 1}.'
'My name is {name}. Next year I will be {age + 1}.'
```

Useful String Methods

- Several string methods analyze strings or create transformed string values.

The upper(), lower(), isupper(), and islower() String Methods

- The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

- These methods do not change the string itself but return new string values.
- If we want to change the original string, we have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored.
- The upper() and lower() methods are helpful if we need to make a case-insensitive comparison.
- In the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.
- **Program**

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

Output

```
How are you?
GREat
I feel great too.
```

- The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns False.

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

- Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well. Expressions that do this will look like a chain of method calls.

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX String Methods

- There are several string methods that have names beginning with the word is. These methods return a Boolean value that describes the nature of the string.
- Here are some common isX string methods:
    - **isalpha()** returns True if the string consists only of letters and is not blank.
    - **isalnum()** returns True if the string consists only of letters and numbers and is not blank.
    - **isdecimal()** returns True if the string consists only of numeric characters and is not blank.
    - **isspace()** returns True if the string consists only of spaces, tabs, and newlines and is not blank.
    - **istitle()** returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

- The isX string methods are helpful when you need to validate user input.
- For example, the following program repeatedly asks users for their age and a password until they provide valid input.
- **Program**

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

output

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

The startswith() and endswith() String Methods

- The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

- These methods are useful alternatives to the == equals operator if we need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() String Methods

Join()

- The join() method is useful when we have a list of strings that need to be joined together into a single string value.

- The join() method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list.

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

- string join() calls on is inserted between each string of the list argument.
    - **Ex:** when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.
    - join() is called on a string value and is passed a list value.

Split()

- The split() method is called on a string value and returns a list of strings.

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

- We can pass a delimiter string to the split() method to specify a different string to split upon.

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

- A common use of split() is to split a multiline string along the newline characters.

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

- Passing split() the argument '\n' lets us split the multiline string stored in spam along the newlines and return a list in which each item corresponds to one line of the string.

Splitting Strings with the partition() Method

The partition() string method can split a string into the text before and after a separator string. This method searches the string it is called on for the separator string it is passed, and returns a tuple of three substrings for the "before," "separator," and "after" substrings. Enter the following into the interactive shell:

```
>>> 'Hello, world!'.partition('w')
('Hello, ', 'w', 'orld!')
>>> 'Hello, world!'.partition('world')
('Hello, ', 'world', '!')
```

If the separator string you pass to partition() occurs multiple times in the string that partition() calls on, the method splits the string only on the first occurrence:

```
>>> 'Hello, world!'.partition('o')
('Hell', 'o', ', world!')
```

If the separator string can't be found, the first string returned in the tuple will be the entire string, and the other two strings will be empty:

```
>>> 'Hello, world!'.partition('XYZ')
('Hello, world!', '', '')
```

You can use the multiple assignment trick to assign the three returned strings to three variables:

```
>>> before, sep, after = 'Hello, world!'.partition(' ')
>>> before
'Hello,'
>>> after
'world!'
```

The partition() method is useful for splitting a string whenever you need the parts before, including, and after a particular separator string.

Justifying Text with rjust(), ljust(), and center()

- The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- The **first** argument to both methods is an integer length for the justified string.

```
'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello, World'.rjust(20)
'         Hello, World'
>>> 'Hello'.ljust(10)
'Hello
```

- 'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.
- An optional **second** argument to rjust() and ljust() will specify a fill character other than a space character.

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

- The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

- These methods are especially useful when you need to print tabular data that has the correct spacing.
- In the below program, we define a printPicnic() method that will take in a dictionary of information and use center(), ljust(), and rjust() to display that information in a neatly aligned table-like format.
- The dictionary that we'll pass to printPicnic() is picnicItems.
- In picnicItems, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

Program

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

output

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.....     4
apples.............    12
cups................     4
cookies.............  8000
```

Removing Whitespace with strip(), rstrip(), and lstrip()

- The strip() string method will return a new string without any whitespace characters at the beginning or end.
- The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

```
>>> spam = '    Hello World    '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World    '
>>> spam.rstrip()
'    Hello World'
```

- Optionally, a string argument will specify which characters on the ends should be stripped.

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

- Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam.
- The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').