Manipulating Strings

1. Introduction

2. String formatting

3. String method

Text is one of the most common forms of data our programs will handle. We already know how to concatenate two `string` together with the `+` operator, but we can do much more than that!

Text is one of the most common forms of data our programs will handle. We already know how to concatenate two `string` together with the `+` operator, but we can do much more than that!

We can extract partial strings from `string` just like sequence, add or remove spacing, convert letters to lowercase or uppercase, and check that `strings` are formatted correctly!

String

There are several ways to create a new `string`; the simplest is to enclose the elements in single, double or triple quotes:

There are several ways to create a new `string`; the simplest is to enclose the elements in single, double or triple quotes:

In [2]:
```
type(''), type(""), type("""""")
```

Out[2]:
```
(str, str, str)
```

There are several ways to create a new `string`; the simplest is to enclose the elements in single, double or triple quotes:

In [2]:
```
type(''), type(""), type("""""")
```

Out[2]:
```
 (str, str, str)
```

In [3]:
```
print("I'am fine")
```

```
I'am fine
```

There are several ways to create a new `string`; the simplest is to enclose the elements in single, double or triple quotes:

In [2]:
```
type(''), type(""), type("""""")
```

Out[2]:
```
(str, str, str)
```

In [3]:
```
print("I'am fine")
```

```
I'am fine
```

A `string` is a **sequence that maps index to case sensitive characters and thus belongs to sequence data type**. Anything that we can apply to the sequence can also be applied to `string`. For instance, we can access the **elements (characters)** one at a time with the bracket operator.

There are several ways to create a new `string`; the simplest is to enclose the elements in single, double or triple quotes:

In [2]:
```
type(''), type(""), type("""""")
```

Out[2]:
```
(str, str, str)
```

In [3]:
```
print("I'am fine")
```

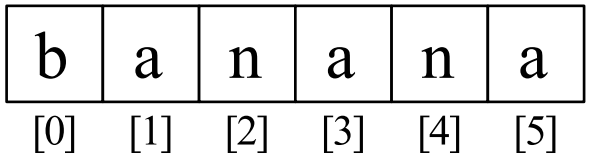
```
I'am fine
```

A `string` is a **sequence that maps index to case sensitive characters and thus belongs to sequence data type**. Anything that we can apply to the sequence can also be applied to `string`. For instance, we can access the **elements (characters)** one at a time with the bracket operator.

In [4]:
```
fruit = 'banana'
fruit[1]
```

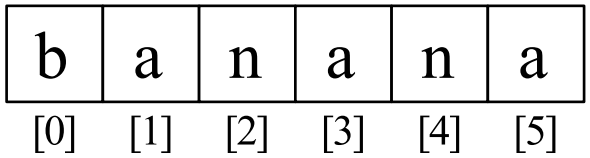
Out[4]:
```
'a'
```

So "b" is the 0th letter ("zero-th") of "banana", "a" is the 1th letter ("one-th"), and "n" is the 2th ("two-th") letter.



source: https://www.py4e.com/html3/06-strings

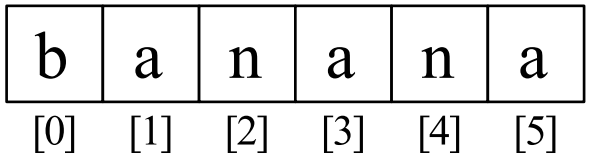
So "b" is the 0th letter ("zero-th") of "banana", "a" is the 1th letter ("one-th"), and "n" is the 2th ("two-th") letter.



source: https://www.py4e.com/html3/06-strings

`len()` can be used to return the number of characters in a `string`:

So "b" is the 0th letter ("zero-th") of "banana", "a" is the 1th letter ("one-th"), and "n" is the 2th ("two-th") letter.



source: https://www.py4e.com/html3/06-strings

`len()` can be used to return the number of characters in a `string`:

In [5]:
```
len(fruit)
```

Out[5]:
```
6
```

We can use negative indices, which count backward from the end of the string.

We can use negative indices, which count backward from the end of the string.

In [6]:
```
fruit[-1], fruit[-2]
```

Out[6]:
```
('a', 'n')
```

We can use negative indices, which count backward from the end of the string.

In [6]:
```
fruit[-1], fruit[-2]
```

Out[6]:
```
('a', 'n')
```

Slicing also works on `string` to extract a substring from the original string. Remember that we can slice sequences using `[start:stop:step]`. The operator `[start:stop]` returns the part of the string from the "start-th" character to the "stop-th" character, including the first but excluding the last with `step=1`.

In [7]:
```
s = 'Cool-Python'
print(s[:5]) #same as s[0:5]
print(s[5:]) #same as s[5:len(s)]
print(s[::2]) #same as s[0:len(s):2]
print(s[::]) #same as s[:] and s[0:len(s):1] => copy the string
print(s[::-1]) #same as s[-1:-(len(s)+1):-1] => reverse the string
```

```
Cool-
Python
Co-yhn
Cool-Python
nohtyP-looC
```

In [7]:
```
s = 'Cool-Python'
print(s[:5]) #same as s[0:5]
print(s[5:]) #same as s[5:len(s)]
print(s[::2]) #same as s[0:len(s):2]
print(s[::]) #same as s[:] and s[0:len(s):1] => copy the string
print(s[::-1]) #same as s[-1:-(len(s)+1):-1] => reverse the string
```

```
Cool-
Python
Co-yhn
Cool-Python
nohtyP-looC
```

In [8]:
```
s = "hello" # `Strings` are "immutable", which means that it cannot be modifi
s[0] = 'y'
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_6820\2526582181.py in <module>
      1 s = "hello" # `Strings` are "immutable", which means that it c
annot be modified
----> 2 s[0] = 'y'

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

The "object" in this case, is the `string` and the "item" is the character we tried to assign. The best we can do is create a new `string` that is a variation on the original:

The "object" in this case, is the `string` and the "item" is the character we tried to assign. The best we can do is create a new `string` that is a variation on the original:

In [9]:
```
print(id(s))
s = 'y' + s[1:len(s)]
print(id(s))
print(s)
```

```
2544442647920
2544446710832
yello
```

A lot of computations involve processing a `string` one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. The traversal of `string` is just like we see before:

A lot of computations involve processing a `string` one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. The traversal of `string` is just like we see before:

In [10]:
```
# Test if s contains 'o'
b = False
for char in s: # Retrieve item (character) one by one
    if char == 'o':
        b = True
        break
print(b)
```

```
True
```

A lot of computations involve processing a `string` one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. The traversal of `string` is just like we see before:

In [10]:
```
# Test if s contains 'o'
b = False
for char in s: # Retrieve item (character) one by one
    if char == 'o':
        b = True
        break
print(b)
```

```
True
```

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

A lot of computations involve processing a `string` one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. The traversal of `string` is just like we see before:

In [10]:
```
# Test if s contains 'o'
b = False
for char in s: # Retrieve item (character) one by one
    if char == 'o':
        b = True
        break
print(b)
```

```
True
```

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

In [11]:
```
print('o' in 'Hello')
print('cats' not in 'cats and dogs')
```

```
True
False
```

In [12]:
```
display_quiz(path+"string1.json", max_width=800)
```

What is printed by the following statements?

```
s = "python rocks"
print(s[2] + s[-4])
```

- tr
- to
- ps
- Error, you cannot use the [ ] operator with the + operator.

Escape Characters

An ***escape character*** lets us use characters that are otherwise impossible to put into a `string`. An escape character consists of a backslash (`\`) followed by the character we want to add to the string.

An ***escape character*** lets us use characters that are otherwise impossible to put into a `string`. An escape character consists of a backslash (`\`) followed by the character we want to add to the string.

For example, the escape character for a single quote is `\'`. We can use this inside a string that begins and ends with single quotes.

An ***escape character*** lets us use characters that are otherwise impossible to put into a `string`. An escape character consists of a backslash (`\`) followed by the character we want to add to the string.

For example, the escape character for a single quote is `\'`. We can use this inside a string that begins and ends with single quotes.

In [13]:
```
spam = 'Say hi to Bob\'s mother.'
spam
```

Out[13]:
```
"Say hi to Bob's mother."
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the `string`. The escape characters `\'` and `\"` let us put single quotes and double quotes inside our strings, respectively.

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the `string`. The escape characters `\'` and `\"` let us put single quotes and double quotes inside our strings, respectively.

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

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the `string`. The escape characters `\'` and `\"` let us put single quotes and double quotes inside our strings, respectively.

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

In [14]:
```
print("Hello there!\nHow are you?\n\tI\'m doing fine.")
```

```
Hello there!
How are you?
        I'm doing fine.
```

In [15]:
```
display_quiz(path+"escape.json", max_width=800)
```

What is printed by the following statement?

```
print("Hello\nWorld!\tWelcome to \"Python\" programming.\\")
```

```
Hello\nWorld!\tWelcome to \"Python\" programmi
ng.\\
```

```
Hello\nWorld!   Welcome to "Python" programmi
ng.\
```

```
Hello
World!   Welcome to "Python" programmi
ng.\
```

```
Hello
World!\tWelcome to "Python" programmi
ng.\
```

> Exercise 1: Ask LLM about the concept of escape character and list some examples

- ChatGPT
- Gemini
- Copilot

Refer to https://hackmd.io/@phonchi/LLM_Tutor

Raw Strings

We 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`.

We 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`.

In [16]:
```
print(r'That is Carol\'s cat.')
```

```
That is Carol\'s cat.
```

We 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`.

In [16]:
```
print(r'That is Carol\'s cat.')
```

```
That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the `string` and not as the start of an escape character.

We 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`.

In [16]:
```
print(r'That is Carol\'s cat.')
```

```
That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the `string` and not as the start of an escape character.

Raw strings are helpful if we are typing strings that contain many backslashes, such as the `strings` used for Windows file paths like `r'C:\Users\Al\Desktop'`.

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:

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:

In [17]:
```
name = 'AI'
age = 33
language = 'Python'
print("Hey! I'm " + name + ", " + str(age)+ " old and I love " + language + "
```

```
Hey! I'm AI, 33 old and I love Python Programing
```

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:

In [17]:
```
name = 'AI'
age = 33
language = 'Python'
print("Hey! I'm " + name + ", " + str(age)+ " old and I love " + language + "
```

```
Hey! I'm AI, 33 old and I love Python Programing
```

However, this requires a lot of tedious typing. A simpler approach is to use ***string interpolation***. The format operator, `%` allows us to construct `strings`, replacing parts of the `strings` with the data stored in variables.

The first operand is the ***format string***, which contains one or more ***format specifiers*** that specify how the second operand is formatted. The result is again a `string`.

The first operand is the ***format string***, which contains one or more ***format specifiers*** that specify how the second operand is formatted. The result is again a `string`.

For example, the format specifiers `%d` means that the second operand should be formatted as an integer ("d" stands for "decimal"):

The first operand is the ***format string***, which contains one or more ***format specifiers*** that specify how the second operand is formatted. The result is again a `string`.

For example, the format specifiers `%d` means that the second operand should be formatted as an integer ("d" stands for "decimal"):

source: https://towardsdatascience.com/python-string-interpolation-829e14e1fc75

The first operand is the ***format string***, which contains one or more ***format specifiers*** that specify how the second operand is formatted. The result is again a `string`.

For example, the format specifiers `%d` means that the second operand should be formatted as an integer ("d" stands for "decimal"):

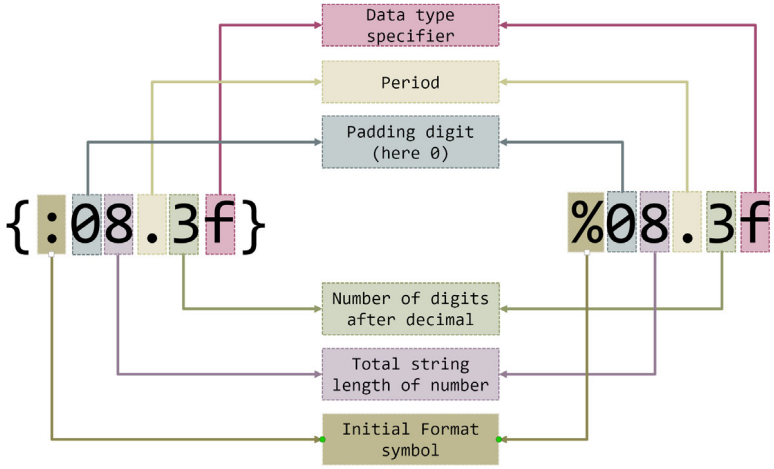
source: https://towardsdatascience.com/python-string-interpolation-829e14e1fc75

In [18]:
```
print("\nHey! I'm %s, %d years old and I love %s Programing"%(name,age,langua
```

```
Hey! I'm AI, 33 years old and I love Python Programing
```

We can have more control over the formatting, for instance:

We can have more control over the formatting, for instance:



source: https://refactored.ai/microcourse/notebook?path=content%2F02-Python_for_Data_Scientists%2F03-Data_Structures_in_python%2F01-Basic_data_types_and_operators.ipynb

In [19]:
```
a = 32
b = 32.145
print('a=%4d, b=%6.2f' % (a,b))
```

```
a=  32, b= 32.15
```

In [19]:
```
a = 32
b = 32.145
print('a=%4d, b=%6.2f' % (a,b))
```

```
a=  32, b= 32.15
```

In [20]:
```
a = 32
b = 32.145
print(f'a={a:4d}, b={b:6.2f}')
```

```
a=  32, b= 32.15
```

Python 3.6 introduced ***f-strings*** (The `f` is for format), which is similar to string interpolation except that braces are used instead of `%`, with the variables or expressions placed directly inside the braces.

Python 3.6 introduced ***f-strings*** (The `f` is for format), which is similar to string interpolation except that braces are used instead of `%`, with the variables or expressions placed directly inside the braces.

Like raw strings, f-strings have an `f` prefix before the starting quotation mark.

Python 3.6 introduced ***f-strings*** (The `f` is for format), which is similar to string interpolation except that braces are used instead of `%`, with the variables or expressions placed directly inside the braces.

Like raw strings, f-strings have an `f` prefix before the starting quotation mark.

In [21]:
```
print(f"\nHey! I'm {name}, {age+2} years old and I love {language} Programing
```

```
Hey! I'm AI, 35 years old and I love Python Programing
```

We can have more control with the f-string besides the field width, like specifying left, right and center alignment with `<`, `>` and `^`. Note now the format specifiers are placed after the variable separated by a colon:

We can have more control with the f-string besides the field width, like specifying left, right and center alignment with `<`, `>` and `^`. Note now the format specifiers are placed after the variable separated by a colon:

In [22]:
```
print(f'[{a:<15d}]') # a = 32, b = 32.145
print(f'[{b:^9.2f}]')
```

```
[32              ]
[  32.15  ]
```

We can have more control with the f-string besides the field width, like specifying left, right and center alignment with `<`, `>` and `^`. Note now the format specifiers are placed after the variable separated by a colon:

In [22]:
```
print(f'[{a:<15d}]') # a = 32, b = 32.145
print(f'[{b:^9.2f}]')
```

```
[32              ]
[  32.15  ]
```

In addition, we can use `+` before the field width specifies that a positive number should be preceded by a `+`. A negative number always starts with a `-`. To fill the remaining characters of the field with 0s rather than spaces, place a `0` before the field width (and after the `+` if there is one):

We can have more control with the f-string besides the field width, like specifying left, right and center alignment with `<`, `>` and `^`. Note now the format specifiers are placed after the variable separated by a colon:

In [22]:
```
print(f'[{a:<15d}]') # a = 32, b = 32.145
print(f'[{b:^9.2f}]')
```

```
[32              ]
[  32.15  ]
```

In addition, we can use `+` before the field width specifies that a positive number should be preceded by a `+`. A negative number always starts with a `-`. To fill the remaining characters of the field with 0s rather than spaces, place a `0` before the field width (and after the `+` if there is one):

In [23]:
```
print(f'[{a:+10d}]')
print(f'[{a:+010d}]')
```

```
[       +32]
[+000000032]
```

In [24]:
```
display_quiz(path+"string2.json", max_width=800)
```

What is printed by the following statements?

```
name = "Alice"
age = 30
balance = 1234.5678
print(f"{name} is {age} years old and has a balance of {balance:.2f}")
```

- Alice is 30 years old and has a balance of 1234.57
- name is 30 years old and has a balance of 1234.57
- Alice is 30 years old and has a balance of 1234.5678
- Alice is {age} years old and has a balance of {balance:.2f}

Exercise 2: Assuming we are designing a word game called "The Mysterious Island" and we need to print the statistics of the player each time the game begins. Try to complete the following function that receives the variables from the game and displays the information that right aligns with each other using the f-string:

```
Player1 Stats:
Health:           100/100
Gold:               0/150
Experience:   50.00/60.00


Player2 Stats:
Health:            60/100
Gold:             120/150
Experience:   40.00/60.00
```

Hint: The maximal width required for each row is 25.

In [ ]:

```
def print_stats(player_name, health, experience, gold):
    print(f"{player_name} Stats:")
    print(f"Health:{____:____}/100")
    print(f"Gold:{____:____}/150")
    print(f"Experience:{____:____}/60.00")
```

In [ ]:

```
game_title = "The Mysterious Island"

welcome_message = f'Welcome to "{game_title}" adventure!\n\n'
# 1. Print the welcome_message
print(welcome_message)

# 2. Use string and number formatting to print out the statistics
player_name = "Player1"
health = 100
gold = 0
experience = 50.000

print_stats(player_name, health, experience, gold)

print("\n")

player_name = "Player2"
health = 60
gold = 120
experience = 40.0

print_stats(player_name, health, experience, gold)
```

String method

`Strings` are an example of Python ***objects***. An object contains both data (the actual `string` itself) and methods, which are effective functions that are built into the object and are available to any instance of the object.

`Strings` are an example of Python ***objects***. An object contains both data (the actual `string` itself) and methods, which are effective functions that are built into the object and are available to any instance of the object.

Python has a function called `dir()`, which lists the methods available for an object.

`Strings` are an example of Python ***objects***. An object contains both data (the actual `string` itself) and methods, which are effective functions that are built into the object and are available to any instance of the object.

Python has a function called `dir()`, which lists the methods available for an object.

In [25]:
```
print(dir(s))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__
doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getit
em__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_sub
class__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul
__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__sub
classhook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'is
alnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isuppe
r', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'rem
oveprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rp
artition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 's
trip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

The `upper()`, `lower()` 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:

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:

In [26]:
```
spam = 'Hello, world!'
spam = spam.upper()
print(spam)
spam = spam.lower()
print(spam)
```

```
HELLO, WORLD!
hello, world!
```

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:

In [26]:
```
spam = 'Hello, world!'
spam = spam.upper()
print(spam)
spam = spam.lower()
print(spam)
```

```
HELLO, WORLD!
hello, world!
```

Note that 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()` string methods return a **new** **`string`** where all the letters in the original `string` have been converted to uppercase or lowercase:

In [26]:
```
spam = 'Hello, world!'
spam = spam.upper()
print(spam)
spam = spam.lower()
print(spam)
```

```
HELLO, WORLD!
hello, world!
```

Note that 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!

This is why we must use `spam = spam.upper()` to change the string in spam instead of simply `spam.upper()`.

The `upper()` and `lower()` methods are helpful if we need **to make a case-insensitive comparison**. For example, the strings `'great'` and `'GREat'` are not equal to each other. But 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.

The `upper()` and `lower()` methods are helpful if we need **to make a case-insensitive comparison**. For example, the strings `'great'` and `'GREat'` are not equal to each other. But 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.

In [27]:
```
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.')
```

```
How are you?

I hope the rest of your day is good.
```

In [28]:
```
display_quiz(path+"string3.json", max_width=800)
```

What is printed by the following statements:

```
s = "ball"
r = ""
for item in s:
    r = item.upper() + r
print(r)
```

- Ball
- BALL
- LLAB

The `isX()` Methods

There are several other `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:

There are several other `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:

- `isupper()/islower()` Returns `True` if the string has at least one letter and all the letters are uppercase or lowercase
- `isalpha()` Returns `True` if the string consists only of letters and isn't 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 and is not blank

In [29]:
```
print('Hello, world!'.islower())
print('hello, world!'.islower())
print('hello'.isalpha())
print('hello123'.isalnum())
print('hello123'.isdecimal())
print(' '.isspace())
print('This Is Title Case'.istitle())
```

```
False
True
True
True
False
True
True
```

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:

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:

In [31]:
```
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.')
```

```
Enter your age:
32
Select a new password (letters and numbers only):
23er
```

In [32]:
```
display_quiz(path+"string4.json", max_width=800)
```

What is printed by the following statements?

```
s1 = "Hello World"
s2 = "Hello123"
s3 = "12345"
s4 = "Hello 123"
s5 = ""

print(s1.isalpha(), s2.isalnum(), s3.isdecimal(), s4.isalnum(), s5.isalpha())
```

```
False, True, False, False, False
```

```
True, False, True, False, True
```

```
True, True, True, False, False
```

```
True, True, True, True, False
```

The `replace()` methods

The `replace()` function is like a "search and replace" operation in a word processor:

The `replace()` function is like a "search and replace" operation in a word processor:

In [34]:
```
greet = 'Hello Bob'
nstr = greet.replace('Bob','Jane')
print(nstr)
```

```
Hello Jane
```

The `join()` and `split()` Methods

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

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

In [35]:
```
print(', '.join(['cats', 'rats', 'bats']))     #Separated by comma
print(' '.join(['My', 'name', 'is', 'Simon'])) #Separated by white space
```

```
cats, rats, bats
My name is Simon
```

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

In [35]:
```
print(', '.join(['cats', 'rats', 'bats']))     #Separated by comma
print(' '.join(['My', 'name', 'is', 'Simon'])) #Separated by white space
```

```
cats, rats, bats
My name is Simon
```

In [36]:
```
' and '.join(['cats', 'rats', 'bats'])
```

Out[36]:
```
'cats and rats and bats'
```

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

In [35]:
```
print(', '.join(['cats', 'rats', 'bats']))     #Separated by comma
print(' '.join(['My', 'name', 'is', 'Simon'])) #Separated by white space
```

```
cats, rats, bats
My name is Simon
```

In [36]:
```
' and '.join(['cats', 'rats', 'bats'])
```

Out[36]:
```
'cats and rats and bats'
```

Notice that the string `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is `'cats, rats, bats'`.

The `split()` method does the opposite: It's called on a string and returns a list of strings.

The `split()` method does the opposite: It's called on a string and returns a list of strings.

In [37]:
```
'My name is Simon'.split()
```

Out[37]:
```
['My', 'name', 'is', 'Simon']
```

The `split()` method does the opposite: It's called on a string and returns a list of strings.

In [37]:
```
'My name is Simon'.split()
```

Out[37]:
```
['My', 'name', 'is', 'Simon']
```

By default, the `string` 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list.

The `split()` method does the opposite: It's called on a string and returns a list of strings.

In [37]:
```
'My name is Simon'.split()
```

Out[37]:
```
['My', 'name', 'is', 'Simon']
```

By default, the `string` 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list.

You can pass a delimiter string to the `split()` method to specify a different string to split upon:

The `split()` method does the opposite: It's called on a string and returns a list of strings.

In [37]:
```
'My name is Simon'.split()
```

Out[37]:
```
['My', 'name', 'is', 'Simon']
```

By default, the `string` 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list.

You can pass a delimiter string to the `split()` method to specify a different string to split upon:

In [38]:
```
'cats, rats, bats'.split(',')
```

Out[38]:
```
['cats', ' rats', ' bats']
```

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

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

In [39]:
```
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')
```

Out[39]:
```
['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']
```

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

In [39]:
```
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')
```

Out[39]:
```
['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`.

Removing Whitespace with the `strip()`, `lstrip()` and `rstrip()` Methods

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. 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.

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. 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.

In [40]:
```
spam = '    Hello, World    \n'
spam.strip()
```

Out[40]:
```
'Hello, World'
```

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. 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.

In [40]:
```
spam = '    Hello, World    \n'
spam.strip()
```

Out[40]:
```
'Hello, World'
```

In [41]:
```
spam.lstrip()
```

Out[41]:
```
'Hello, World    \n'
```

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. 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.

In [40]:
```
spam = '    Hello, World    \n'
spam.strip()
```

Out[40]:
```
'Hello, World'
```

In [41]:
```
spam.lstrip()
```

Out[41]:
```
'Hello, World    \n'
```

In [42]:
```
spam.rstrip()
```

Out[42]:
```
'    Hello, World'
```

Exercise 3: When editing the markdown document, you can create a bulleted list by putting each list item on its own line and placing a `-` in front. But say you have a really large list to which you want to add bullet points. You could just type those `-` at the beginning of each line, one by one. Or you could automate this task with a short Python program! For example, if I have following text:

```
Lists of resources
Lists of books
Lists of videos
Lists of blogs
```

After running the program, the text should contain the following:

```
- Lists of resources
- Lists of books
- Lists of videos
- Lists of blogs
```

In [ ]:

```
text = """Lists of resources
Lists of books
Lists of videos
Lists of blogs"""

# 1. Separate lines into list using string method.
lines = text._____("\n")

# 2. Add -
for i, line in enumerate(lines):    # loop through all indexes for "lines" li
    lines[i] = _______              # add - to each string in "lines" list

# 3. Use string method to conctenate list of strings back to string
text = "\n"._______(lines)
print(text)
```

In [43]:
```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch6.json')
```

string

Next

>