Lecture 5

January 28, 2023

1 Introduction to strings

- A **string** is a sequence of characters enclosed in quotes (either single or double).
- In Python, strings are immutable, meaning that once created, the elements within a string cannot be changed or replaced.
- Strings are a built-in data type and have many built-in methods that can be used to manipulate and analyze them.

1.1 Creating Strings

Strings can be created by enclosing a sequence of characters in quotes (either single or double).

```
string1 = "Hello, world!"
string2 = 'Hello, world!'
```

You can also use the **str()** function to convert a non-string data type (such as a number) into a string.

```
num = 42
string3 = str(num)
```

To create a string with multiple lines, you can use triple quotes (either single or double).

```
string4 = """This is a
multi-line string."""
```

1.2 String Indexing and Slicing

- Strings are ordered sequences, and each element in the sequence can be accessed using an index.
- The indexing starts from 0.

```
string1 = "Hello, world!"
print(string1[0]) # 'H'
print(string1[-1]) # '!'
```

You can also use slicing to access a range of characters within a string

```
string1 = "Hello, world!"
print(string1[7:12]) # 'world'
```

1.3 String Methods:

- Python strings have many built-in methods that can be used to manipulate and analyze them.
- Some of the most commonly used methods include:
- **len()**: Returns the length of a string.
- **lower()**: Returns the string in lowercase.
- **upper()**: Returns the string in uppercase.
- **str.replace(old, new)**: Replaces all occurrences of the old string with the new string.
- **str.split(sep)**: Splits the string into a list of substrings using the specified separator.
- **str.join(iterable)**: Joins all the items in an iterable, such as a list or tuple, into a single string.

[ ]:
```
string1 = "Hello beautiful world!"
print(len(string1))
print(string1.upper())
print(string1.replace('world', 'Math 183'))
print(string1.split(' '))
```

1.4 String Concatenation and Repetition

You can concatenate strings using the "+" operator.

```
string1 = "Hello"
string2 = "world"
string3 = string1 + ', ' + string2 + '!'
print(string3) # 'Hello, world!'
```

You can also repeat a string multiple times using the "*" operator.

```
string1 = "Hello"
string2 = string1 * 3
print(string2) # 'HelloHelloHello'
```

2 Formatting

- The **format()** method in Python is used to insert placeholders, represented by curly braces {}, into a string and then replace them with the corresponding values.
- The placeholders can be replaced by positional arguments, where the first curly brace is replaced by the first argument, the second curly brace is replaced by the second argument, and so on.

[ ]:
```
# format() method with positional arguments:

name = 'Linh'
age = 20
```

```
print('My name is {} and I am {} years old.'.format(name, age))
```

2.1 Keyword arguments

- you can replace the placeholders with keyword arguments, where each placeholder is replaced by the value of the corresponding key.

[2]:
```
# format() method with keyword arguments
kevin_name = 'Kevin'
kevin_age = 40
print('My name is {name} and I am {age} years old.'.format(name=kevin_name,␣
 ↪age=kevin_age))
```

```
My name is Kevin and I am 40 years old.
```

2.2 f-string

You can use the f-strings or formatted string literals, which is a more readable and concise way of formatting strings introduced in Python 3.6 and later versions.

```
name = 'John'
age = 30
print(f'My name is {name} and I am {age} years old.')
```

You could also use the % operator for formatting strings.

```
name = 'John'
age = 30
print('My name is %s and I am %d years old.' % (name, age))
```

Practice problem Write a function that takes the name and age from the user. Then print out their self introduction: "Hello, my name is … and I am …years old".

[5]:
```
def intro():
    name = input('Input your name:')
    age = input('Input your age:')
    age = int(age)
    print("My name is %s and I am %d years old" %(name,age))
intro()
```

```
Input your name:Linh
Input your age:22
My name is Linh and I am 22 years old
```

3 Advanced practice problems

1. Palindrome Checker: Write a function that takes in a string and checks if it is a palindrome (a word, phrase, or sequence that reads the same backwards as forwards).

2. Anagram Checker: Write a function that takes in two strings and checks if they are anagrams (words or phrases formed by rearranging the letters of a different word or phrase).
3. Reverse String: Write a function that takes in a string and reverses it.
4. Reverse Words: Write a function that takes in a sentence and reverses the order of the words while maintaining the order of the letters within each word.
5. Find the Longest Word: Write a function that takes in a sentence and returns the longest word in it.
6. Character Count: Write a function that takes in a string and returns a dictionary of the count of each character in the string.
7. Text Justification: Write a function that takes in a string and a maximum line width and returns the string justified to the given width.
8. String Permutations: Write a function that takes in a string and generates all possible permutations of the characters in the string.
9. String Compression: Write a function that takes in a string and compresses it by counting the number of consecutive occurrences of each character and returning the compressed string.
10. Snake Case: Write a function that takes in a string and converts it to snake case (a string where words are separated by underscores and all letters are in lowercase).

4 Regular expressions

Regular expressions also known as "regex" or "regexp", are a powerful tool for matching patterns in strings. They are used in Python to search, edit, and manipulate text.

Here are some key concepts and methods for working with regular expressions in Python:

- **re** module: built-in module that provides functions for working with regular expressions.
- **search()** method: This method searches for a match to a regular expression in a given string. It returns a match object if a match is found, or None if no match is found.
- **findall()** method: This method finds all non-overlapping matches to a regular expression in a given string and returns them as a list of strings.
- **sub()** method: This method replaces all occurrences of a regular expression in a given string with a replacement string.
- **compile()** method: This method is used to create a regular expression pattern object, which can be used for multiple searches.

[8]:
```
import re

# search
text = "The quick brown fox jumps over the lazy dog"
match = re.search(r'fox', text)
print('search result:', match.start(), match.end())
```

```python
# findall
text = "The quick brown fox jumps over the lazy dog"
matches = re.findall(r'\w+', text)
print('findall result:',matches)

# sub
text = "The quick brown fox jumps over the lazy dog"
new_text = re.sub(r'fox', 'cat', text)
print('sub result:', new_text)

# compile
text = "The quick brown fox jumps  over the lazy dog"
pattern = re.compile(r'fox')

match = pattern.search(text)
print('compile result:', match.start(), match.end())

text2 = "The fox jumps  over the lazy dog"
match = pattern.search(text2)
print('compile result:', match.start(), match.end())
```

```
search result: 16 19
findall result: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog']
sub result: The quick brown cat jumps over the lazy dog
compile result: 16 19
compile result: 4 7
```

4.1 Special characters

Regular expressions use special characters to represent patterns. For example:

- “.”: represents any character. Example: **.a** represents any string with two characters, the first letter can be anything, the second letter is “a”.
- “*“: represents zero or more occurrences of the preceding character. Example: ___a*___ can be ““,”a”,“aa”, etc.
- “+”: represents one or more occurrences of the preceding character. Example: **a+** represent “a”, “aa”, “aaa”, etc.
- “?”: Matches zero or one occurrence of the preceding character or group. For example, **a?** matches zero or one occurrence of the letter “a”.
- “{n}”: represents exactly n occurrences of the preceding character or group. For example, **a{3}** matches exactly three occurrences of the letter “a”.
- “{n,}”: represents n or more occurrences of the preceding character or group. For example, **a{3,}** matches three or more occurrences of the letter “a”.

- "{n,m}": represents at least n and at most m occurrences of the preceding character or group. For example, **a{2,4}** matches two, three, or four occurrences of the letter "a".
- "[]": Matches any one of the characters inside the square brackets. For example: **[abc]** represents any one of the letters "a", "b", or "c".
- "^": represents the start of a line or string. For example, **^a** means the letter "a" at the start of a line or string.
- "$": represents the end of a line or string. For example, **a$** matches the letter "a" at the end of a line or string.
- "|": means either the expression before or the expression after the vertical bar. For example, a|b matches either the letter "a" or the letter "b".
- "": Escapes special characters to match them literally. For example, "" matches the period character "." literally.

These are just a few examples of special characters in regular expressions. There are many more special characters and advanced techniques that can be used to create complex regular expressions that can match a wide variety of patterns.

Practice: (a) Write a regular expression that matches any valid phone number in the format (xxx) xxx-xxxx

(b) Write down a program to check if a phone number is in the valid format (xxx) xxx-xxxx

[10]:
```
import re

def is_valid_phone(phone):
    pattern = r"^\(\d{3}\) \d{3}-\d{4}$"
    if re.match(pattern, phone):
        return True
    return False

phone = "(13) 456-7890"
if is_valid_phone(phone):
    print(f"{phone} is a valid phone number")
else:
    print(f"{phone} is not a valid phone number")
```

```
(13) 456-7890 is not a valid phone number
```

Problem: write down a program to check whether an email address is valid using regular expression

[3]:
```
import re

def is_valid_email(email):
    pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    if re.match(pattern, email):
        return True
    return False
```

```
email = "someone@uidaho.edu"
if is_valid_email(email):
    print(f"{email} is a valid email address")
else:
    print(f"{email} is not a valid email address")
```

```
someone@uidaho.edu is a valid email address
```

5 Check out ticket

Problem: Given a string that contains a phone number in the format "XXX-XXX-XXXX", where X is a digit, write a regular expression and a Python script that extracts the phone number from the string and prints it to the console.