STATS 507
Data Analysis in Python

Week2-1: Strings, Iteration, and Lists.

Dr. Xian Zhang

Recap: primitive data type in Python

Different **object** can **represent** different concepts.

ANY object has a type that defines what kind of operations programs can do to them

- int, -- represent integers, ex: 507
- float, -- represent real numbers, ex: 3.1415, 2.0
- bool, -- represent Boolean values, ex: True, False
- NoneType -- special and has one value, None

Mathematical operator: +, -, *, /, **, //, %

Logical operator: and, or, not

Recap: variable and expressions

Variables

```
In [1]: mystring = "It has been a lovely day."
        approx_pi = 3.1415
        number_of_planets = 9
```

Variable is a name that **refers** to a **value.**

Assign a value to a variable via assignment operator “=”

Expressions

Combine objects and operators to form expressions.

- (507 * 12) / 3

```
<object> <operator> <object>
```

Mathematical, Boolean and Conditional Expressions

```
if x > 0:
    if x < 10:
        print('x is a positive single-digit number')
```

```
if 0 < x and x < 10:
    print('x is a positive single-digit number')
```

Recap -- Functions

Calling functions in Python (built-in or from another module)

```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.4142135623730951
```

1. We have to **import** (i.e., load) the related module (if needed)
2. Call function

Defining functions in Python

```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

```
Welcome to Python programming
Let's start with function definition
```

1. Create new function using function definition
2. Call the newly defined function

Besides basic primitive structures…

1. Strings in Python

2. Iteration

3. Lists in Python

Strings in Python

String is an **immutable** sequence of case sensitive characters.

- Letters, special characters, spaces, digits
- “me”, ‘States 507'
- Another built-in **date type** in Python

```
lecture_intro = "Lecture 2 talks about string"
type(lecture_intro)
```

```
str
```

Create a string (single or double quote)

- str1 = “This is a string”
- str2 = ‘This is also be a string’

https://docs.python.org/3/library/stdtypes.html#str

What operations can we do with strings?

Concatenation using “+”

```
a = "Hello"
b = "world!"
c = a + " " + b
print(c)
```

```
Hello world!
```

Repeat using “*”

```
a = "Hello"
c = a * 3
print(c)
```

```
HelloHelloHello
```

Getting the length of a string using len()

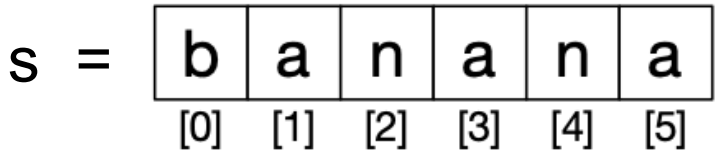
len() is a function used to retrieve the length of a string in the parantheses

All Python sequences include a length attribute, which is the number of elements in the sequence

Operations with strings: Indexing

Indexing into the string to get the value at a certain index/position by using square brackets.

Indexing performed by square brackets. Python sequences are **0-indexed**. The index counts the offset from the beginning of the sequence. So the first letter is the 0-th character of the string.




```
s = "banana"
```

```
print(s[0], s[1], s[2], s[3], s[4], s[5])
```

```
b a n a n a
```

What will happen?.

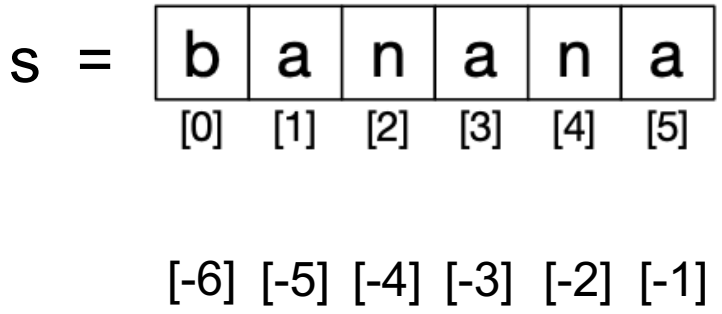
```
s[6]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 s[6]

IndexError: string index out of range
```

Negative Indexing

We can index intro a string with **negative** index as well.




```
s = "banana"
```

```
print(s[-1], s[-2], s[-3], s[-4], s[-5], s[-6])
```

```
a n a n a b
```

IndexErrror if we go too far to the left…

```
s[-7]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[13], line 1
----> 1 s[-7]

IndexError: string index out of range
```

Slicing

We can **slice** strings to get a substring using `[start:stop:step]`

Get a sequence/subsequence of characters at start up to and including stop -1 taking every step characters.

- If [start: stop], step = 1 by default
- If [:], picks out the entire string
- Looking at the step first

+ ⟶

- ⟵

s =

b	a	n	a	n	a
[0]	[1]	[2]	[3]	[4]	[5]

```
s[1:5]      #anan
s[1:5:2]    #aa
s[:]        #banana
s[5:1:-2]   #aa
```

What can we do with strings -- comparison

Sometimes we want to check if two strings are equal

```
1 'cat' == 'cat'
```

```
True
```

```
1 'cat' == 'dog'
```

```
False
```

```
1 'dog' == 'doge'
```

```
False
```

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

If we can compare strings with equality, we should be able to compare them with inequalities, too...

Comparison

We can also compare words under alphabetical ordering.

```
1 'cat' < 'dog'
```

```
True
```

```
1 'cat' >= 'dog'
```

```
False
```

```
1 'dog' < 'doge'
```

```
True
```

```
1 '' < 'goat'
```

```
True
```

```
1 '1' < 'a'
```

```
True
```

Words earlier in the dictionary are “smaller” than words later in the dictionary.

The empty string `''` comes first in the ordering.

Strings including numbers, symbols, etc. are also ordered.

Comparison

Note: upper case and lower case letters ordered differently

```
1 'Cat' == 'cat'
```

```
False
```

```
1 'cat' > 'Cat'
```

```
True
```

Upper case letters are ordered before lower case letters.

For more information:
https://docs.python.org/3/library/stdtypes.html#comparisons

For **much** more information:
https://docs.python.org/3/library/operator.html?highlight=equallity

In-class practice with string

Immutability

Python strings are immutable -> can not be modified

```
s = "string"
s[1] = "p"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 s = "string"
----> 2 s[1] = "p"

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

Can create a new object that assign the same variable name

```
s = "string"
s = s[0] + 'p' + s[2:]
s
```

```
'spring'
```

This avoids the error we saw. It changes the value of the variable s by essentially creating a new string, rather than trying to change the content of a string.

Other Python string methods

Python strings provide a number of built-in operations, called **methods**

A **method** is like a function, but it is provided by an **object**. We'll learn much more about this later in the semester, but for now, it suffices to know that some data types provide what *look* like functions (they take arguments and return values), and we call these function-like things **methods**.

```
1 mystr = 'goat'
2 mystr.upper()
```

```
'GOAT'
```

```
1 'aBcDeFg'.lower()
```

```
'abcdefg'
```

```
1 'banana'.find('na')
```

```
2
```

```
1 'goat'.startswith('go')
```

```
True
```

This variable.method() notation is called **dot notation**, and it is ubiquitous in Python (and many other languages).

https://docs.python.org/3/library/stdtypes.html#string-methods

Two more useful things (dir, help) for string (showcase in jupyter…)

Optional arguments: `str.find()`

Optional arguments in Python are function parameters that have a default value. This allows you to call the function without specifying those arguments, in which case the default values are used.

```
1 'banana'.find('na')
```

```
2
```

```
1 'banana'.find('na', 3)
```

```
4
```

```
1 'banana'.find('na', 3, 4)
```

```
-1
```

```
1 'banana'.find('na', 3, 6)
```

```
4
```

The `str.find()` method takes **optional arguments**, which specify where in the string to start looking for a match, and the last index to consider for a match.

Find first occurrence of `'na'`, starting from index `3`.

Find first occurrence of `'na'`, starting from index `3`, and nowhere past `4`.

The documentation writes this method as `str.find(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`. Square brackets indicate optional arguments. In this case, brackets also indicate that with two arguments, the second one will be interpreted as the `start` argument.
https://docs.python.org/3/library/stdtypes.html#string-methods

Searching sequence: the in keyword

```
1 'a' in 'banana'
```

```
True
```

```
1 'z' in 'banana'
```

```
False
```

```
1 'ban' in 'banana'
```

```
True
```

```
1 'anan' in 'banana'
```

```
True
```

```
1 'zoo' in 'banana'
```

```
False
```

`x in y` returns `True` if `x` occurs as a substring of `y`, and `False` otherwise.

Importantly, we can check for a whole substring, making this very similar to `str.find()`.

The **`in`** keyword applies more generally to check whether an object is contained in a sequence. We'll see more examples of this in the future, but for now, we only need to worry about strings.

1. Strings in Python

2. Iteration

3. Lists in Python

Iteration

Why?

Quite often, we find ourselves to run the same bit of code over and over again.

How?

Iterative algorithm use **while loops and for loops** to

- for loop: run over over element of some set
- while loop: repeat until some conditions is met

While loop

One form of iteration in Python is the `while` statement

The iterative algorithm using a `while` loop. As long as the conditional expresson evaluates to be `True`, Python will run the code in the body of the loop and re-check the condition.

```
5]: n = 0
    while n < 5:
        print(n)
        n += 1
    print("while loop DONE")
```

```
0
1
2
3
4
while loop DONE
```

Set loop variable outside of the `while` loop

Test loop variable in condition
Do something…
Modify the loop variable within the `while` loop

While loop (infinite loop…)

Note: one should always try to ensure that a while loop (eventually terminate). Make sure the condition will eventually be evaluated to be false.

```
def countdown(n):
    while n > 0:
        print(n)
        # n = n - 1
    print("We have lift off")
```

Warning: There is a danger of creating an **infinite loop**. If, for example, `n` never gets updated, then when we call `countdown(10)` , the condition `n>0` will always evaluate to `True`, and we will never exit the while-loop.

While loop: the `break` keyword

We can also terminate a while-loop using the `break` keyword

```
1 a = 4
2 x = 3.5
3 epsilon = 10**-6
4 while True:
5     print(x)
6     y = (x + a/x)/2
7     if abs(x-y) < epsilon:
8         break
9     x=y # update to our new estimate
```

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

The `break` keyword terminates the current loop when it is called.

Note: this is an implementation of Newton_Raphson method: https://en.wikipedia.org/wiki/Newton's_method which you will get to learn more in your HW.

While loop: the `continue` keyword

We can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
> Done
Done
>      While
     While
> # Do not print
> done
Done!
```

The `continue` keyword finish the current iteration by doing nothing and immediately jump to the next iteration.

Iteration: **for** loop for repeated action

Another form of iteration in Python is the `for` statement

```
5]: n = 0
    while n < 5:
        print(n)
        n += 1
    print("while loop DONE")
```

```
0
1
2
3
4
while loop DONE
```

```
for i in range(5):
    print(i)
print("for loop DONE")
```

```
0
1
2
3
4
for loop DONE
```

We use `for` statement when looping through a know set/list of items
We call the `while` statement until some **conditions** becomes false

Recursion

Recursion is anther way to repeat instructions. In code, recursion is implemented using a function that calls itself.

```
def countdown(n):
    if n <= 0:
        print('We have lift off!')
    else:
        print(n)
        countdown(n-1)
```

Countdown calls itself!

But the key is that each time it calls itself, it is passing an argument with its value decreased by 1, so eventually, `n <= 0` is true.

```
1 countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Infinite Recursion and RuntimeError

With a small change, we can make it so that `countdown(1)` encounters an **infinite recursion**, in which it repeatedly calls itself

```
def countdown(n):
    if n <= 0:
        print("We have lift off")
    else:
        print(n)
        countdown(n)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[4], line 1
----> 1 countdown(10)

Cell In[3], line 6, in countdown(n)
      4 else:
      5     print(n)
----> 6     countdown(n)

Cell In[3], line 6, in countdown(n)
      4 else:
      5     print(n)
----> 6     countdown(n)

    [... skipping similar frames: countdown at line 6 (2967 times)]

Cell In[3], line 6, in countdown(n)
      4 else:
      5     print(n)
```

Note: a **RecursionError** is a specific type of **RuntimeError** that when a recursive function exceeds the maximum depth of recursion

String Traversal: `for` and `while`

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. This pattern of processing is called a **traversal**.

```
s = "banana"
for c in s:
    print(c)
```

```
b
a
n
a
n
a
```

For-loop and the in keywords provides a more concise way to traverse the string.

```
s = "banana"
i = 0
while i < len(s):
    print(s[i])
    i = i + 1
```

```
b
a
n
a
n
a
```

In-class practice

1. Strings in Python

2. Iteration

3. Lists in Python

Lists in Python

Strings in Python are “sequences of characters”

But what if I want a sequence of something else?

- A vector would be naturally represented as a sequence of numbers
- A class roster might be represented as a sequence of strings

Lists are sequences whose values can be of **any data type**

- We call those list entries the **elements** of the list

Creating Lists

We create(construct) a list by putting its elements between **square brackets**, separated by commas.

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
3 mixed = ['one', 2, 3.0]
4 pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

This is a list of four strings

This is a list of nine integers

The elements of a list need not be same type. Here is a list with a string, an integer and a float.

A list can even contain more lists! This is a list of three lists, each of which is a list of three integers.

Creating an empty list

It is possible to construct a list with no elements, the empty list.

Here are two equivalent ways of creating an empty list.

```
1 x = []
2 x
```

```
[]
```

Creating a list using square brackets notation, but supply no elements, x is empty

```
1 x = list()
2 x
```

```
[]
```

Use the reserved keyword `list`, which casts to a list, given no argument. It constructs an empty list.

Lists concatenation: + and *

List concatenation is similar to string concatenation

```
1 fibonacci = [0,1,1,2,3,5,8]
2 primes = [2,3,5,7,11,13]
3 fibonacci + primes
```

```
[0, 1, 1, 2, 3, 5, 8, 2, 3, 5, 7, 11, 13]
```

```
1 3*['cat','dog']
```

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

These operations are precisely analogous to the corresponding string operations. This makes sense, since both strings and lists are **sequences**.
https://docs.python.org/3/library/stdtypes.html#typesseq

Lists have length: `len()`

Lists are sequences, so they have a length

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 len(fruits)
```

```
4
```

```
1 len([])
```

```
0
```

The empty list have length 0, just like the empty string

```
1 pythagoras = [[3, 4, 5], [5, 12, 13], [8, 15, 17]]
2 len(pythagoras)
```

```
3
```

Note: one might be tempted to say that `Pythagoras` have length 9, but each element of a list counts only once, even if it is itself a more complicated object.

Indexing

We can access individual elements of a list just like a string. This is because both strings and lists are examples of Python **sequences**.

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 fruits[0]
```

```
'apple'
```

```
1 fruits[1]
```

```
'orange'
```

```
1 fruits[2]
```

```
'banana'
```

```
1 fruits[-1]
```

```
'kiwi'
```

Can also index from the end of the list, just like with strings.

Slicing

Exactly like string, We can **slice** lists using `[start:stop:step]`

Get a sequence/subsequence of characters at start up to and including stop -1 taking every step characters.

```
l = [0,1,2,3,4,5,6]
l[:]       #[0, 1, 2, 3, 4, 5, 6]
l[0:5]     #[0, 1, 2, 3, 4, 5]
l[0:5:2]   #[0, 2, 4]
l[:3]      #[0, 1, 2]
l[3:]      #[3, 4, 5, 6]
l[-1]      # 6
l[::-1]    #?
l[4:1:-2] #?
```

Lists are mutable

Unlike strings, lists are mutable. We can change individual elements after creating the list.

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 fruits
```

```
['apple', 'orange', 'banana', 'kiwi']
```

```
1 fruits[-1] = 'mango'
2 fruits
```

```
['apple', 'orange', 'banana', 'mango']
```

```
1 mystring = 'goat'
2 mystring[0]='b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-b526da741b9a> in <module>()
      1 mystring = 'goat'
----> 2 mystring[0]='b'

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

Reminder of what happens if we try to do this with a string. This error is because string are **immutable**. Once they're created, they can't be altered.

Other things

HW1 due this week.

HW2 out today.

Coming next:

More on lists (methods), set, dictionaries…