STATS 507
Data Analysis in Python

Week3-1: Lists, Tuples, Files

Dr. Xian Zhang

About slicing…

We can **slice** a string using `[start:stop:step]`

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 if the indices are out of bounds?

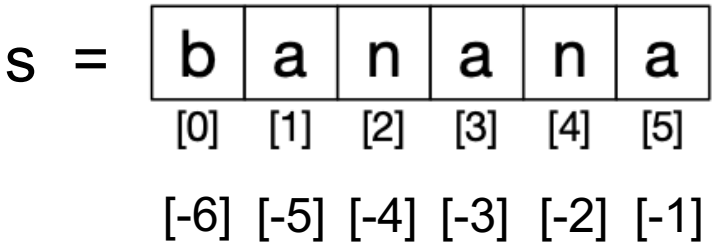
Out-of-bounds indices are adjusted to sequence bounds

```
s = 'banana'
s[1:10:2]      # "aaa"      (stop 10 becomes len(s)=6)
s[8:20:2]      # ""         (start > len(s), empty result)
s[len(s)::-1]  # "ananab"   (negative step, start becomes len(s) - 1)
```

From Python documents: link

Recap: Strings operations in Python

We can **index** intro a string using `[]`.



We can **slice** a string using `[start:stop:step]`

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

Other string methods

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

```
'GOAT'
```

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

```
'abcdefg'
```

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

```
2
```

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

```
True
```

Recap: Iteration & Recursion

Why?

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

- `while` loop: loop as long as a condition is `True`
- `for` loop: loop variable takes on values in a sequence
- Recursive: function calls for itself until end condition is met

In-class practice solution:

```
def sum_naturals_while(n):
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

```
def sum_naturals_recur(n):
    if n <= 1:
        return n
    else:
        return n + sum_naturals_recur(n-1)
```

Recap -- Lists

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

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

Create a list (denoted by) `[]` or `list()`

```
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]]
```

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

```
[]
```

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

```
[]
```

Sequence: indexing, slicing, `len()` …

1. Lists in Python

2. Tuples in Python

3. Files in Python

Since Lists are mutable…

How can we mutate a list?

Mutable: We can **change** values of specific elements of a list.

Add new element, **delete** existing ones, **reorder(sort)** and many more…

Assign to an element at an index **changes** the list value

```
l = [1,2,3]
print(l, id(l))
l[1] = 5
print(l, id(l))
```

```
[1, 2, 3] 5098075264
[1, 5, 3] 5098075264
```

Lists operations -- `append()`

We call list methods with **dot notation** (talked in last lecture). These are methods supported by certain **objects** (a concept we will cover later in class)

Add an element to **end** of the list with `L.append(element)`

list object

method name

method argument

Mutate the original list – to be one element longer.

```
animals = ['cat', 'dog', 'goat', 'bird']
print(animals, id(animals))

# Append will mutate the original list
animals.append('wolverine')
print(animals, id(animals))
```

```
['cat', 'dog', 'goat', 'bird'] 5098007488
['cat', 'dog', 'goat', 'bird', 'wolverine'] 5098007488
```

```
animals = animals.append('wolverine') # animals evaluates to be?
```

```
print(animals)
```

```
None
```

Be careful! The `append` operation does a mutation, but **returns** a None object as a result…

Lists operations -- `insert()`

Add an element to **a specific location** of the list with

```
L.insert(idx, element)
```

Mutate the original list and does NOT return anything.

```
animals = ['cat', 'dog', 'goat', 'bird']
print(animals, id(animals))

# Insert element to a specific location.
animals.insert(0,'wolverine')
print(animals, id(animals))
```

```
['cat', 'dog', 'goat', 'bird'] 5099578176
['wolverine', 'cat', 'dog', 'goat', 'bird'] 5099578176
```

Lists Operation – `extend()`

Add **multiple** element to the list. `L.extend(another_list)`

```
animals = ['cat', 'dog', 'goat', 'bird']
print(animals, id(animals))

# extend can add multiple values.
animals_2 = ['wolverine', 'fish']
animals.extend(animals_2)
print(animals, id(animals))
```

```
['cat', 'dog', 'goat', 'bird'] 5099579648
['cat', 'dog', 'goat', 'bird', 'wolverine', 'fish'] 5099579648
```

Another function that add elements and mutates the list and **return nothing**

We use these functions their side effects.

Question: How is this different from concatenation by “+” operator?

`list.append()` and `extend()`

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 animals.append('unicorn')
3 animals
```

```
['cat', 'dog', 'goat', 'bird', 'unicorn']
```

```
1 fibonacci = [0,1,1,2,3,5,8]
2 fibonacci.append([13,21])
3 fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, [13, 21]]
```

Warning: `list.append()` adds its argument as the last element of a list! Use `list.extend()` to concatenate to the end of the list!

```
1 fibonacci = [0,1,1,2,3,5,8]
2 fibonacci.extend([13, 21])
3 fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Note: all of these list methods act upon the list that calls the method. These methods don't return the new list, they alter the list on which we call them.

Lists Operation `list.remove()`

Removes the first instance of x in the list by `list.remove(element)`.

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 animals.remove('cat')
3 animals
```

```
['dog', 'goat', 'bird']
```

```
1 numbers = [0,1,2,3,1,2,3,2,3]
2 numbers.remove(2)
3 numbers
```

```
[0, 1, 3, 1, 2, 3, 2, 3]
```

`list.remove(x)` removes the first instance of `x` in the list.

```
1 numbers.remove(4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-160-6d289ee6c03d> in <module>()
----> 1 numbers.remove(4)

ValueError: list.remove(x): x not in list
```

Raises a `ValueError` if no such element exists.

Lists Operation `list.pop()`

`list.pop()` does two things:1) remove the last element from the list (mutate)

2) return that element

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 animals.pop()
```

```
'bird'
```

`list.pop()` removes the last element from the list and returns that element.

```
1 animals
```

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

```
1 fibonacci = [0,1,1,2,3,5,8]
2 fibonacci.pop(3)
```

```
2
```

`list.pop()` takes an **optional argument**, which indexes into the list and removes and returns the indexed element

```
1 fibonacci
```

```
[0, 1, 1, 3, 5, 8]
```

Again, this method alters the list itself, rather than returning an altered list.

Lists Operation – reorder…

Sort a list.

`list.sort()` is a **method** associated with list and sorts the list **in place**. See documentation for how Python sorts data of different types: https://docs.python.org/3/howto/sorting.html

- l = [5, 0,7]
- l.sort() -> [0, 5, 7]
- Mutates the list

Sorted()

`sorted(l)` returns a sorted version of a list, leaving its argument unchanged

- l = [5, 0,7]
- l_new – sorted(l)
- No mutation, original list unchanged.

Reverse a list.

- Reverse the list
- Mutate the list

```
: l = [5, 0, 7]
print(l)
l.reverse()
print(l)
```

```
[5, 0, 7]
[7, 0, 5]
```

Strings to Lists

Convert string to list with `list(s)`.

- Every character from s is an element in list

```
my_str = "List() can cast string"
print(list(my_str))
```

```
['L', 'i', 's', 't', '(', ')', ' ', 'c', 'a', 'n', ' ', 'c', 'a',
's', 't', ' ', 's', 't', 'r', 'i', 'n', 'g']
```

Use `s.split(char)` to split a string on a character parameter, splits on spaces if called without a parameter.

```
l1 = my_str.split(' ')
print('l1:', l1)

l2 = my_str.split('(')
print('l2:', l2)
```

```
l1: ['List()', 'can', 'cast', 'string']
l2: ['List', ') can cast string']
```

Lists to Strings

Convert a list of strings back to string

Use `'char'.joint(L)` to return a list of strings into a bigger string.

Can give a character/strings in quotes to insert between each given string.

```
l = ['xianzhang', 'Desktop', 'Stats507']

s1 = ''.join(l)
print(s1)
```

```
xianzhangDesktopStats507
```

```
s2 = '/'.join(l)
print(s2)
```

```
xianzhang/Desktop/Stats507
```

Iterating over a List

Similarly to string, list supports iteration.

```
nums = [5,0,7]

# Using range and len
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Can also iterate over list elements directly

```
nums = [5, 0, 7]
for n in nums:
    print(n)
```

```
5
0
7
```

List Comprehensions

Using list comprehensions

```
[x/f(x) for x in my_list if boolean_expr]
```

output expression sequence filter

List comprehension is a concise and powerful way to create new lists based on existing lists or other iterable objects.

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 [x.upper() for x in animals if 'o' in x[1]]
```

```
['DOG', 'GOAT']
```

In-class practice

1. Lists in Python

2. Tuples in Python

3. Files in Python

What are tuples…

Recall:

Strings are **immutable** sequences of case sensitive characters.

Lists are sequences whose values can be of any data type

Tuples are **immutable** sequences whose values can be of **any data type**

Creating tuples…

Create a tuple: tuples are created with “comma notation”, with optional parentheses:

- tuple_1 = 1, 2, 3
- tuple_2 = (2,3)
- tuple_3 = (2, "UM week 3", 98.0, True, [1,2,3])

Creating a tuple of **one** element requires a trailing comma.

```
t = ("cat")
type(t)
```

```
str
```

```
t = ("cat", )
type(t)
```

```
tuple
```

Can also use the `tuple()` function

Create a tuple using the `tuple()` function

```
t = tuple(("cat", "dog"))
type(t)
```

```
tuple
```

The `tuple()` function behaves similarly to casting functions `int()`…

It can case any sequence to a tuple

```
print(type(range(5)))
t2 = tuple(range(5))
print(t2, type(t2))
```

```
<class 'range'>
(0, 1, 2, 3, 4) <class 'tuple'>
```

```
print(type([1,2,3,4,5]))
t2 = tuple([1,2,3,4,5])
print(t2, type(t2))
```

```
<class 'list'>
(1, 2, 3, 4, 5) <class 'tuple'>
```

```
print(type("string"))
t2 = tuple("string")
print(t2, type(t2))
```

```
<class 'str'>
('s', 't', 'r', 'i', 'n', 'g') <class 'tuple'>
```

Tuples are sequences: index and slice

Indexing performed by square brackets: **0-indexed**.

Slicing sequences (tuples) by: `[start:stop:step]`

```
seq = (2, "UM week 3", 98.0, True, [1,2,3])
seq[0]      # 2
seq[0:3]    # (2, 'UM week 3', 98.0)
seq[3:]     #(True, [1, 2, 3])
seq[4][-1] # 3
seq[5]      # ?
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[89], line 6
      4 seq[3:]    #(True, [1, 2, 3])
      5 seq[4][-1] # 3
----> 6 seq[5]

IndexError: tuple index out of range
```

Tuples are immutable

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-c40a1905a6e9> in <module>()
      1 fruits = ('apple', 'banana', 'orange', 'kiwi')
----> 2 fruits[2] = 'grapefruit'

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

Tuples are immutable so changing an entry is not permitted.

```
1 fruits = fruits[0:2] + ('grapefruit',) + fruits[3:]
2 fruits
```

```
('apple', 'banana', 'grapefruit', 'kiwi')
```

Similar with strings, we have to make a new assignment to the variable.

```
1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-f62749483e65> in <module>()
----> 1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]

TypeError: can only concatenate tuple (not "str") to tuple
```

Note: even though ‘grapefruit’ is a tuple, Python does not now how to parse this lime,. We need **use parenthesis!**

So what do we use tuples for?

Swap variable types.

```
a = 1
b = 2
b = a
a = b
print(a, b)
```

```
1 1
```

```
a = 1
b = 2
temp = a
a = b
b = temp
print(a, b)
```

```
2 1
```

```
a = 1
b = 2
a, b = b, a
print(a, b)
```

Evaluate to be a tuple (2, 1)

```
2 1
```

One at a time, assign value to variables (in a sequence)

Tuple as build-in function return

Return one **tuple** object that consists **multiple** values...

```
# Function return for more than 1 value
t = divmod(5,2)
help(divmod)
```

```
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

```
quotient, remaindar = divmod(5,2)
print(quotient, remaindar)
```

```
2 1
```

https://docs.python.org/3/library/functions.html

Tuple as function return

Tuple can serve as return values for our own defined function. The following function takes a list of numbers and returns a tuple summarizing the list.

```
1 import random
2 def five_numbers(t):
3     t.sort()
4     n = len(t)
5     return (t[0],  t[n//4],  t[n//2], t[(3*n)//4], t[-1])
6 five_numbers([1,2,3,4,5,6,7])
```

```
(1, 2, 4, 6, 7)
```

```
1 randnumlist = [random.randint(1,100) for x in range(60)]
2 (mini,lowq,med,upq,maxi) = five_numbers(randnumlist)
3 (mini,lowq,med,upq,maxi)
```

```
(3, 27, 54, 73, 98)
```

Ref: https://en.wikipedia.org/wiki/Five-number_summary

Tuples as function input

Optional (variable-length) arguments can be bundled into a tuple as input.

```
1 def my_min( *args ):
2     return min(args)
3 my_min(1,2,3)
```

```
1
```

```
1 my_min(4,5,6,10)
```

```
4
```

```
1 def print_all( *args ):
2     print(args)
3 print_all('cat', 'dog', 'bird')
```

```
('cat', 'dog', 'bird')
```

```
1 print_all()
```

```
()
```

When a parameter name in a function definition is prefaced with an asterisk (*), this indicates that it can take any number of positional arguments beyond those already named. **These extra positional arguments, if any, are bundled into a tuple**. This feature is often used to write **flexible** functions that can handle an arbitrary number of arguments.

Note: this is also one of several ways that one can implement **optional arguments**, though we'll see better ways later in the course.

Tuples in practice: assignment

Tuple assignment works so long as the right hand side is **any sequence,** provided the number of variables matches the number of elements on the right.

```
1 email = 'klevin@umich.edu'
2 email.split('@')
```

```
['klevin', 'umich.edu']
```

The `string.split()` method returns a list of strings, obtained by splitting the calling string on the characters in its argument.

```
1 (user,domain) = email.split('@')
2 user
```

```
'klevin'
```

```
1 domain
```

```
'umich.edu'
```

A string is a sequence, so tuple assignment is allowed. Sequence elements are characters, and indeed, x, y and z are assigned to the three characters in the string.

```
1 (x,y,z) = 'cat'
2 print(x, y, z)
```

```
c a t
```

When to use a tuple (v.s. list)?

Use a **tuple** when :

- The set is unlikely to change during execution
- Need to key on a set (i.e., require immutability)
- Want to perform multiple assignment or for use in variable-length arg list

While use a **list** when :

- Length is not known ahead of time and/pr may change during execution (most code your may see will use lists because of its mutability)
- Frequent updates

Other tuple applications: `zip()`

zip() takes multiple iterable objects (like strings, lists, tuples etc.) and returns an **iterator** of tuples, where each tuple contains the elements from all the iterables that are in the same position.

```
1 t1 = ['apple', 'orange', 'banana', 'kiwi']
2 t2 = [1, 2, 3, 4]
3 zip(t1,t2)
```

```
<zip at 0x10c95d5c8>
```

Notice the return is a zip object, which is an **iterator** containing as its elements formed from its arguments:
Ref: https://docs.python.org/3/library/functions.html#zip

```
1 for tup in zip(t1,t2):
2     print(tup)
```

```
('apple', 1)
('orange', 2)
('banana', 3)
('kiwi', 4)
```

Iterators are, in essence, objects that support for-loops. All sequences are iterators. Iterators support, crucially, a method `__next__()`, which returns the "next element". We'll see this in more detail later in the course.

`zip()` in practice

Given arguments of different lengths, `zip()` defaults to the shortest one. `zip()` can take any number of arguments, as long as they are **iterable**. Sequence are iterable.

```
1 for tup in zip(['a','b','c'],[1,2,3,4]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

```
1 for tup in zip(['a','b','c','d'],[1,2,3]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

```
1 for tup in zip([1,2,3],['a','b','c'],'xyz'):
2     print(tup)
```

```
(1, 'a', 'x')
(2, 'b', 'y')
(3, 'c', 'z')
```

Iterables are, essentially, objects that can become iterators. We'll see the distinction later in the course. https://docs.python.org/3/library/stdtypes.html#typeiter

`zip()` in practice

`zip()` is particularly useful for iterating over several lists in parallel.

```
def count_matches(s, t):
    cnt = 0
    for (a, b) in zip(s, t):
        if a == b:
            cnt += 1
    return cnt
```

Note, the results if zip() is an iterator of tuples

```
count_matches([1,2,3,4,5], [1,4,3])
```

```
2
```

Test your understanding: what should this return?

```
count_matches([0,2,6,4,5], [1,2,3])
```

Other tuple applications: `enumerate()`

Read on your own:

https://docs.python.org/3/library/functions.html#enumerate
On dictionary items():
https://docs.python.org/3/tutorial/datastructures.html

1. Lists in Python

2. Tuples in Python

3. Files in Python

Intro to files: what and why

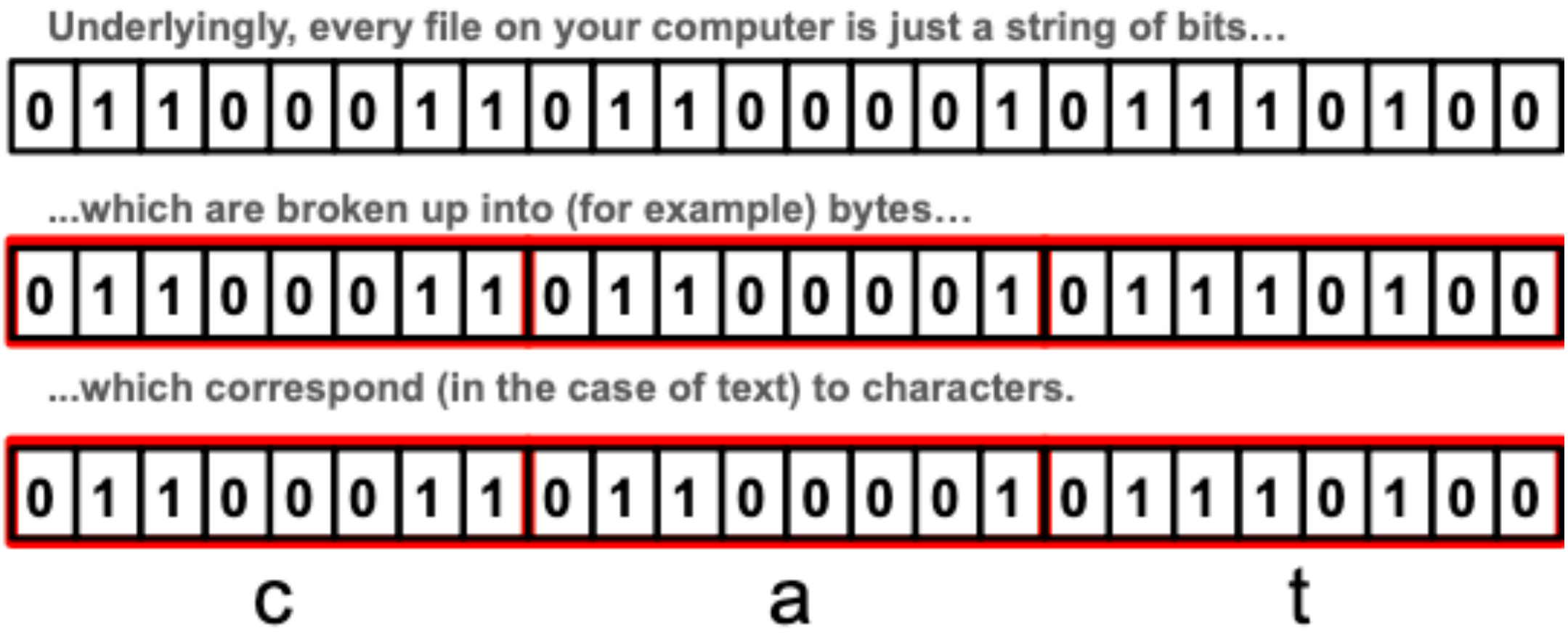
What are files?

- Files are way to **store** and **manage** data on a computer.
- Can contain a wide range of data types, including text, images, music, videos, executable programs, and more
- Are typically organized to **filesystem**: read, write, format and have paths and directories…

Why files are important:

- Persistence, files are available for future access.
- Data exchange: files serve as a common media for data exchange between different programs, systems, or users.
- Easier manage and store: better to organize and scale…

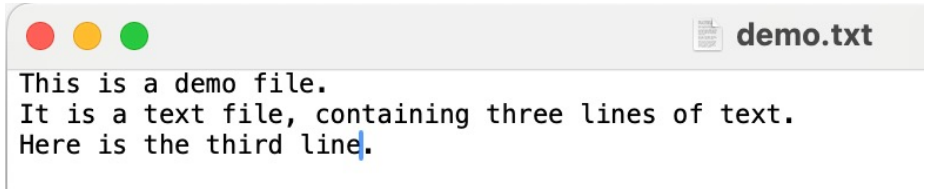
Reading and writing files



x

Reading files in Python: `open()`

`cat` is command line that can print the contents of a file to the screen. But for now, we are going to use the build-in python function `open()`, which allow for more complex file manipulations.




```
$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
$
```

```
1 f = open('demo.txt')
2 type(f)
```

```
_io.TextIOWrapper
```

Open the file `demo.txt`. This creates a **file object** f.
https://docs.python.org/3/glossary.html#term-file-object

```
1 f.readline()
```

```
'This is a demo file.\n'
```

Provide a method for reading a single line from the file. The string '**\n**' is a special character that represents a new line.

Reading files

```
1 f = open('demo.txt')
2 f.readline()
```

```
'This is a demo file.\n'
```

```
1 f.readline()
```

```
'It is a text file, containing three lines of text.\n'
```

```
1 f.readline()
```

```
'Here is the third line.\n'
```

```
1 f.readline()
```

```
''
```

Each time we call `f.readline()`, we get the next line of the file…

… until there are no more lines to read, at which point the `readline()` method return the empty string whenever it is called.

Reading files

We can also open file using the `with` keyword (recommended). `f` can be treated as an iterator, in which each each iteration gives us a line of the file.

```
1 with open('demo.txt') as f:
2     for line in f:
3         for wd in line.split():
4             print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

Iterate over each word in the line (splitting on ' ' by default)

Remove the trailing punctuation from the words of the file.

From the documentation: "It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point." https://docs.python.org/3/reference/compound_stmts.html#with

In plain English: the `with` keyword does a bunch of error checking and cleanup for you, automatically.

`open()` with mode

Files are created with the special function

`open``(filename, `**`mode`**`)`

The mode is tells whether you want the file to used for:

What for: Reading, writing, or appending.

File Type: Text or binary.

```
# text-mode, read-only
open("readme.txt", "rt")
# text mode, write
open("readme.txt", "wt")
# text mode, append
open("readme.txt", "at")
# binary mode, read-only
open("data.dat", "rb")
# binary mode, write
open("data.dat", "wb")
# binary mode, append
open("data.dat", "ab")
```

Writing files in Python

Open the file in **write** mode. If the file already exists, this creates it anew, deleting its old contents.

```
1 f = open('animals.txt', 'w')
2 f.read()
```

If I try to read a file in write mode, I get an error.

```
---------------------------------------------------------------------------
UnsupportedOperation                                 Traceback (most recent call last)
<ipython-input-29-3b1ef477003a> in <module>()
      1 f = open('animals.txt', 'w')
----> 2 f.read()

UnsupportedOperation: not readable
```

Write to the file. This method returns the number of characters written to the file. Note that `'\n'` counts as a single character, the new line.

```
1 f.write('cat\n')
2 f.write('dog\n')
3 f.write('bird\n')
4 f.write('goat\n')
```

```
5
```

Writing files in Python

```
1 f = open('animals.txt', 'w')
2 f.write('cat\n')
3 f.write('dog\n')
4 f.write('bird\n')
5 f.write('goat\n')
6 f.close()
```

Open the file in **write** mode. This overwrites the version of the file created in the previous slide.

Each write appends to the end of the file.

When we're done, we close the file. This happens automatically when the program ends, but its good practice to close the file as soon as you're done.

```
1 f = open('animals.txt', 'r')
2 for line in f:
3     print(line, end="")
```

```
cat
dog
bird
goat
```

Now, when I open the file for reading, I can print out the lines one by one.

The lines of the file already include newlines on the ends, so override Python's default behavior of printing a newline after each line.

Formatting strings in Python

Very commonly, we want to write **formatted** string data to a file.

There are 3 ways of doing this in Python:

- The `%` operator (**old, avoid using this notation**)
- `string.format()`
- `f-strings (newest)`

```
topping = "pineapple"

# all of these print
# "my fav pizza is pineapple"

"my fav pizza is %s" % topping

"my fav pizza is {}".format(topping)
"my fav pizza is {a}".format(a=topping)

f"my fav pizza is {topping}"
```

Formatting strings using %

Python provides tools for formatting strings to make it easier to print integer/ floating point as a string.

```
1 x = 23
2 print('x = %d' % x)
```

```
x = 23
```

```
1 animal = 'unicorn'
2 print('My pet %s' % animal)
```

```
My pet unicorn
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y))
```

```
2.718000 divided by 1.618000 is 1.679852
```

```
1 print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
```

```
2.718 divided by 1.618 is 1.67985167
```

`%d` : integer
`%s` : string
`%f` : floating point
More information:
https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Can further control details of formatting, such as numbers of significant figures in printing floats.

Formatting strings using %

Number of formatting arguments must **match** (no more or no less) the length of the supplied tuple.

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y,1.0))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-eb736fce3612> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y,x/y,1.0))

TypeError: not all arguments converted during string formatting
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b2e6a26d3415> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y))

TypeError: not enough arguments for format string
```

Formatting strings using `string.format()`

The `string.format()` provides a flexible way to create formatted strings.
It uses replacement fields denoted by {} as placeholders for values.

```
x = 2.718
y = 1.618
print("{} divided by {} is {}".format(x, y, x/y))
```

```
2.718 divided by 1.618 is 1.679851668726823
```

We can also control numbers of significant figures in printing floats.

```
x = 2.718
y = 1.618
print("{} divided by {} is {:0.8f}".format(x, y, x/y))
```

```
2.718 divided by 1.618 is 1.67985167
```

Formatting strings using `f-string`

The `F-string`, introduced in Python 3.6, offer a concise and readable way to embed expressions inside string for formatting.

```
x = 2.718
y = 1.618
print(f"{x} divided by {y} is {x/y:0.8f}")
```

```
2.718 divided by 1.618 is 1.67985167
```

What are the advantages?

- Readability: F-strings are more readable and intuitive, especially for complex formatting.
- Performance: F-strings are generally faster than other formatting methods.
- Direct evaluation: You can put any **valid Python expression** inside the curly braces.
- Less error-prone: With `%` and `.format()`, it's easy to mismatch placeholders and arguments.

In-class practice

Other things

HW2 out.

Coming next:

More on files, dictionaries, exceptions.

A note on pace and difficulty:

I aim to teach Python from scratch in this course, but …

…besides basic Python, we also want to cover data science specific tools…Come speak to me if I am moving too fast.