Flulapalooza®

September 28

all faculty, staff, postdoctoral scholars and students are encouraged to attend the event at the area between Light Hall and the VA Hospital to receive their annual flu vaccination

Homework 2

posted on Brightspace
due next Wed (Sep 14)

Homework2.pdf (written description)
Homework2.ipynb (Notebook to use for your solution)

download from Brightspace

`StringsRegularExpressions.ipynb`
`ListsTuples.ipynb`
`SetsAndDictionaries.ipynb`
`Struct.ipynb`
`ListComprehensions.ipynb`

String Types
String Operators
(Continued)

`StringsRegularExpressions.ipynb`

https://docs.python.org/3/tutorial/introduction.html#strings

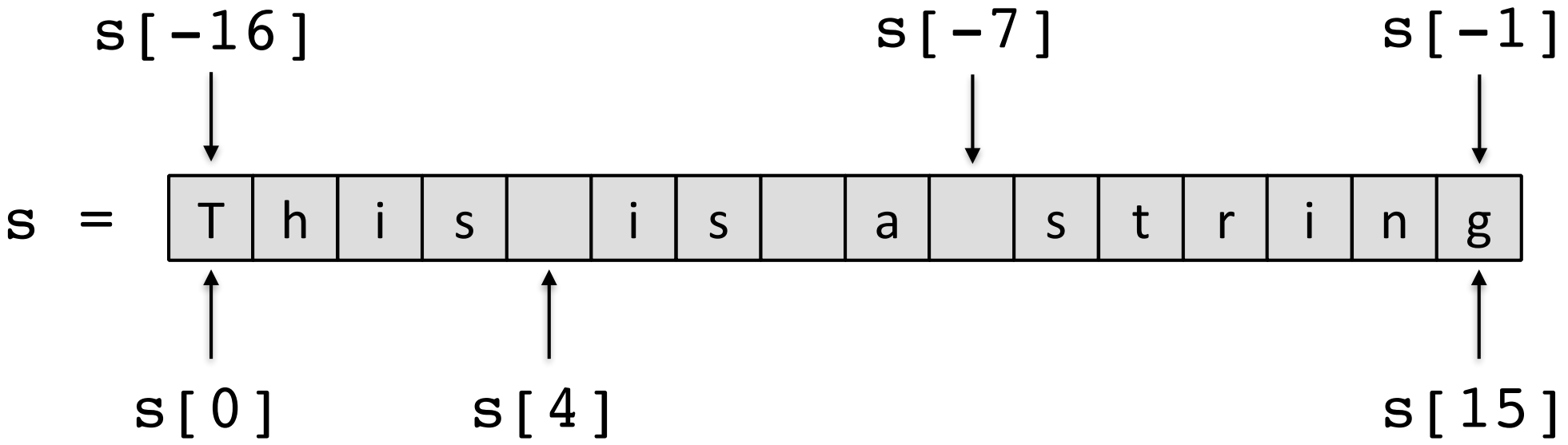
https://docs.python.org/3/library/stdtypes.html#string-methods

string indices

- indices of strings (and lists, tuples, numpy arrays) start at 0, not at 1
- end of a string is index `len(s)-1`
- negative indices count from the end of a string

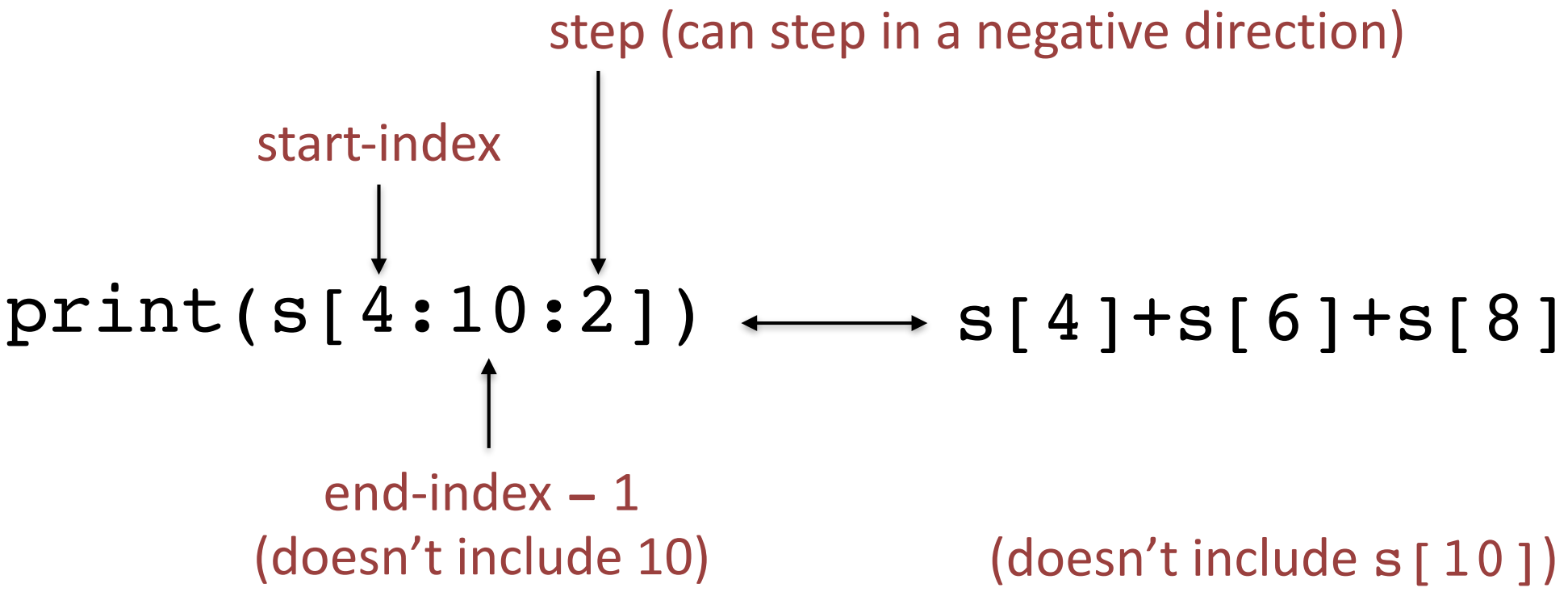
```
s = "This is a string"
print(s[0])             # first char
print(s[1])             # second char
print(s[len(s)-1])      # last char
print(s[-1])            # also last char
```

string indices



string slicing

```
s = "This is a string"
```



stepping through each character in a string introducing **for loops** in Python

```
s = "This is a string"
```

`range()` returns a "sequence" of numbers from 0 to `len(s)-1`

```
for i in range(len(s)):
    print(i, "\t", s[i])
```

colon is necessary

indenting is necessary in Python
must be consistent through a program
convention is to use spaces

automatic in Jupyter Notebooks and IDEs

A Whirlwind Tour of Python, Jake VanderPlas
Chapter 8: Control Flow Statements

stepping through each character in a string introducing **for loops** in Python

```
s = "This is a string"
```

`range()` can have a start, end, and step

```
for i in range(0, len(s), 2):
    print(i, "\t", s[i])
```

a note on `range()`*

* in Python 3, `range()` returns a range object, which is iterable

`range()` does not actually create a list**

** it did in Python 2

```
N = 10 ** 15
```

larger than the memory of any personal computer

```
for i in range(N):
    if i >= 10:
        break
    print(i, end=', ')***
```

*** replaces the default 'new line' with something else, here a `', '`

stepping through each character in a string introducing **for loops** in Python

```
s = "This is a string"
```

can iterate over the string itself

```
for c in s:
    print(c)
```

strings are immutable

```
s = "This is a string"

# cannot change a string
s[3] = "X"   this throws an error
```

strings, tuples

immutable = unchangeable

vs.

mutable = changeable

lists, numpy arrays

(most variables in R are immutable)

Data Structures

Data Structures

one key to successful programming is
using the right kind of data structure and
using it the right way for the right problem

Why use a data structure?

imagine we have 10 subjects and each subject answers 10 true/false questions - 100 data points

we could create 100 variables, s1q1, s2q2, ... s2q1, s2q2, ... s10q9, s10q10

why do we use a data structure instead?

Why use a data structure?

imagine we have 10 subjects and each subject answers 10 true/false questions - 100 data points

we could create 100 variables, s1q1, s2q2, ... s2q1, s2q2, ... s10q9, s10q10

why do we use a data structure instead?

- access data more easily and more efficiently
- access data dynamically
- data are structured, systematically referenced
- all the data is "in the same place"

Data Structures in Python

- List
- Tuple
- Dictionary
- Sets

part of base Python

- Numpy Arrays
- Pandas

imported modules / packages

Python also easily support more sophisticated data structures (stacks, queues, trees, networks, etc.)

- we will use numpy arrays a lot in this course

Lists and Tuples

`ListsTuples.ipynb`

https://docs.python.org/3/tutorial/introduction.html#lists

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

lists [ ]

- declare lists using square brackets

```
a = [1, 5, 10, 12, 4, 6, 2]
```

- reference item in a list

```
a[2]                    indices start at 0
```

- slice a list

```
a[1:5:2]        start, end (not included), step
```

- length of a list

```
len(a)                      # elements
```

tuples ( )

- declare tuples using parentheses

```
a = (1, 5, 10, 12, 4, 6, 2)
```

- reference item in a tuple

```
a[2]                    indices start at 0
```

- slice a tuple

```
a[1:5:2]        start, end (not included), step
```

- length of a tuple

```
len(a)                      # elements
```

lists and tuples

- both can be homogenous or heterogenous

homogenous list (same types)

```
a = [4, 9, 3, 1]
```

```
b = ["fish", 3.1, 1, True]
```

heterogenous list (different types)

lists are often homogenous
"by convention" (not necessary)

lists are like cell arrays { } in Matlab,
not like arrays [ ] in Matlab

homogenous tuple (same types)

```
a = (4, 9, 3, 1)
```

```
b = ("fish", 3.1, 1, True)
```

heterogenous tuple (different types)

lists and tuples

- lists are mutable (can be changed)

```
a = [2, 5, 10, 3, 9]
a[1] = 99
```

- tuples are immutable* (cannot be changed)

```
a = (2, 5, 10, 3, 9)
a[1] = 99
```

throws an error

immutability can be beneficial to ensure that data in the tuple does not change

we will see that:

- tuples are often used to pass multidimensional parameters to functions
- tuples are returned when multiple parameters are returned from functions

* in a language like R, (nearly) everything is immutable

empty tuples, tuples with one element

- empty tuple

```
a = ()
```

parentheses are required

- tuple with one element

```
a = (99,)
```

comma is necessary

- other ways to create tuple

```
a = 1, 5, 9
```

parentheses are optional (included "by convention")

```
a = 99,
```

comma is necessary, but parentheses optional

these work, but it's convention to show parentheses when creating a tuple

empty lists, lists with one element

- empty list

```
a = []
```

brackets are required

- list with one element

```
a = [99]
a = [99,]
```

comma is optional

converting between lists and tuples

- ```
  mylist  = [10, 5, 1]
  mytuple = (11, 6, 0)

  newlist  = tuple(mytuple)
  newtuple = list(mylist)
  ```

- can do this too (from an iterable to list or tuple):

  ```
  a = list(range(10))
  b = tuple(range(10))
  ```
  ```

operations on lists and tuples

- on lists

```
a = [1, 2, 3]
b = [5, 6, 7]
print(a+b)
print(3*a)
```

- on tuples

```
c = (1, 2, 3)
d = (5, 6, 7)
print(c+d)
print(3*c)
```

note that lists and tuples *are not*
to be used for vector/matrix operations
(use numpy arrays, to be discussed soon)

adding and deleting items on a list

- adding item

```
a = [1, 2, 3, 4, 5, 6]
a.append(99)
```

- deleting item(s)

```
del(a[1])
del(a[0:3])
```

- unlike Matlab, you cannot do this

```
a = [1, 2, 3]
a[10] = 72
```

other operations on lists

```
a = [1, 4, 7, "truck", 3, 4]
```

```
del(a[1])
```

deletes item in index position 1

vs.

```
a.remove(“truck")
```

removes first occurrence of truck, otherwise throws an error

other operations on lists

```
a = ["cow","cow", "ball", "kite"]

print(a.count("cow"))
```

number of occurrences

if statement in Python

: required

```
if (a.count("book")):
    print(a.index("book"))
else:
    print("not found!")
```

: required

index of first occurrence

other operations on lists

+ vs. `extend` vs. `append`

```
a = [1, 2, 3, 4, 5]
b = [6, 7, 8]
```

`a+b` extends a with b (concatenate)

`a.extend(b)` extends a with b (concatenate)

`a.append(b)` nests b in a in an appended index position

other operations on lists

```
a = [5, 4, 6, 1, 4, 3, 5]
```

sort method

```
a.sort()
```

sorts (and modifies) list a

sorted() function

```
sorted(a)
```

returns a new list of elements of a sorted

stepping through each item in a list
for loops in Python

```
a = [1, 2, 3, 5, 7, 11]
```

`range()` iterates through a sequence of numbers from 0 to `len(a)-1`

```
for i in range(len(a)):
    print(i, "\t", a[i])
```

colon is necessary

indenting is necessary in Python
must be consistent through a program
convention is to use spaces
automatic in Jupyter Notebooks and IDEs

A Whirlwind Tour of Python, Jake VanderPlas
Chapter 8: Control Flow Statements

stepping through each item in a list
for loops in Python

```
a = [1, 2, 3, 5, 7, 11]
```

```
for elem in a:
    print(elem)
```

iterate through list directly, rather than by indexing

stepping through each item in a tuple
for loops in Python

```
a = (1, 2, 3, 5, 7, 11)
```

`range()` iterates through a sequence of numbers from 0 to `len(a)-1`

```
for i in range(len(a)):
    print(i, "\t", a[i])
```

colon is necessary

indenting is necessary in Python
must be consistent through a program
convention is to use spaces
automatic in Jupyter Notebooks and IDEs

copy of a list

```
a = [5, 4, 6, 1, 4, 3, 5]

b = a
a[2] = 99
print(a == b)

b = a.copy()
b = a[:]
print(a == b)
```

a and b reference the same thing

a and b are different entities

a and b are different entities

nested (multi-dimensional) lists and tuples

lists and tuples contain object

those object can be other lists and tuples (or strings, or floats, or ints, or any other object)

```
a = [[1, 2, 3], [4, 5, 6]]
print(a[0][2])
```

we will see multidimensional homogenous lists (or tuples) used to create multidimensional (homogenous) numpy arrays later

but of course lists and tuples do not need to be homogeneous

nested (multi-dimensional) lists and tuples

```
a = (1, [2, 3], (4, [5, 6]), 7)
```

```
print(a[1][1])
```
returns 3

```
print(a[0][1])
```
throws error

```
print(a[2][1])
```
returns `[5, 6]`

```
print(a[2][1][1])
```
returns 6

nested (multi-dimensional) lists and tuples

```
a = ([1, 'go'], ((2, (3, 4)), 5))
```

```
print(a[1])
```
returns `((2, (3, 4)), 5)`

```
print(a[0][1][0])
```
returns `'g'`

```
print(a[1][1])
```
returns `5`

```
print(a[1][0][1])
```
returns `(3, 4)`

```
print(a[1][0][1][1])
```
returns `4`

shallow vs. deep copy

see `ListsTuples.ipynb`

review these this Jupyter notebook and code on your own

Sets and Dictionaries

`SetsAndDictionaries.ipynb`

useful in many situations
(also important to recognize them in Python code)

Sets and Dictionaries

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

https://docs.python.org/3/tutorial/datastructures.html#sets

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

sets

sets are unordered
each element is unique
duplicate elements not allowed

sets and dictionaries use curly brackets

```
a = {'r', 'b', 'o', 'b', 'y'}
```

```
len(a)
```
elements in set

```
a.add('p')
```
add element to set

```
a.remove('r')
```
remove element from set

```
a & b
```
intersection

```
a | b
```
union

dictionaries

collection of key-value pairs

duplicate keys are not allowed

```
resp = {'L' : 3, 'M' : 6, 'H' : 3}
resp['L'] += 1

for k in resp:
    print(k, ':', resp[k])
```

Structures

Python does not have a `struct` type

C and Matlab have `struct`, Java does not
R has dataframes (like `pandas` in Python)

`Struct.ipynb`

Python does not have a `struct` type

using dictionaries

```
subj = {}
subj['firstname'] = 'Sally'
subj['lastname'] = 'Jones'
subj['age'] = 63
subj['BDI'] = 38

print(subj)
print(subj['lastname'])
```

this doesn't create a consistent "type" - fine for a one-off struct, bad if you wanted an array of structs

Python does not have a `struct` type

using classes (template for an object)

```
class subject:
    firstname = ''; lastname = ''; age = 999; BDI = 999

    def __init__(self, fname, lname, age, BDI):
        self.firstname = fname
        self.lastname = lname
        self.age = age
        self.BDI = BDI

    def __str__(self):
        return 'subject(firstname='+self.firstname+',
                        lastname='+self.lastname+', age='+str(self.age)+',
                        BDI='+str(self.BDI)+')'

subj = subject('Sally', 'Jones', 63, 38)
print(subj)
print(subj.lastname)
```

a `struct` in Python would be a class with attributes only, and no methods

Python does not have a `struct` type

using namedtuple

```
from collections import namedtuple


subject = namedtuple('subject', ['firstname', 'lastname',
'age', 'BDI'])
subj = subject(firstname='Sally', lastname='Jones',
age=63, BDI=38)
print(subj)
print(subj.lastname)
```

https://docs.python.org/3/library/collections.html#collections.namedtuple

List Comprehensions

`ListComprehensions.ipynb`

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

list comprehensions

this

```
a = [x**2 for x in range(10)]
```

is equivalent to

```
a = []
for x in range(10):
    a.append(x**2)
```

list comprehensions

this

```
b = [(x, y) for x in range(5) for y in (range(5)) if x != y]
```

is equivalent to

```
b = []
for x in range(5):
    for y in range(5):
        if x != y:
            b.append((x, y))
```

list comprehensions

```
[(x, y) for x in range(5) for y in (range(5)) if x != y]
```

While list comprehensions are powerful and potentially more readable for those fluent in Python, they can be very difficult to comprehend for those who are not programming in Python daily.

More verbose code is often easier to understand, especially for those unfamiliar with Python, and easier to debug.

I have seen numerous homework assignments using list comprehensions and other forms of compact code that look nice but are completely wrong (and difficult to unpack to see why they are wrong) - be careful