

Today: string in, if/elif, string .find(), slices

See chapters in Guide: [String](#) - [If](#)

String in Test

- String **in** - test if a substring appears in a string
- Chars must match exactly - recall "case sensitive"
- This is boolean True/False test
use .find() (below) to know **where** substring is
- Mnemonic: Python re-uses the word "in" from the for loop
- General: we'll use in later for other data types
- Strategy: don't write code for something that Python has built-in

```
>>> 'Dog' in 'CatDogBird'
True
>>> 'dog' in 'CatDogBird'      # upper vs. lower
case
False
>>> 'd' in 'CatDogBird'        # finds d at the
end
True
>>> 'atD' in 'CatDogBird'      # not picky about
words
True
>>>
>>> 'x' in 'CatDogBird'
False
```

Variant: not in

There's also a `not in` form which is `True` if the element is not in there, like `!=`. Use this, say for an if-statement where you want to take an action if something is not in a string.

```
>>> s = 'CatDogBird'
>>> if 'x' not in s:           # YES this way
    print('no x')
no x
>>>
>>> if not 'x' in s:         # NO works but not
PEP8
    print('no x')
no x
>>>
```

Using `not in` is preferred for this case vs. the form `not 'x' in s`

Example: `has_pi()`

`has_pi(s)`: Given a string `s`, return `True` if it contains the substrings `'3'` and `'14'` somewhere within it, but not necessarily together. Use `"in"`.

> [`has_pi\(\)`](#)

Note these functions are in the [string-3 section](#) on the Experimental server

boolean-test AND boolean-test

This form looks right in English but does not work in Python or most computer languages:

```
if '3' and '14' in s:      # NO does not work
    ...
```

The **and** should connect two fully formed boolean tests, such as you would write with `"in"` or `"=="`, so this works

```
if '3' in s and '14' in s:  
    ...
```

Strategy: Built In Functions

Python has many built-in functions, like "in", and we will see all the important ones in CS106A. You want to know the common built-in functions, since using a built-in is far preferable to writing code for it yourself - "in" is a nice example. The "in" operator works for several data structure to see if a value is in there, and its use with strings is our first example of it. It also can in some cases run faster than what you could code yourself.

Later Practice: `has_first()`

> [`has_first\(\)`](#).

Example: `catty()`

> [`catty\(\)`](#).

`'xaCtxyzAx' -> 'aCtA'`

Return a string made of the chars from the original string, whenever the chars are one of 'c' 'a' 't', (not case sensitive).

Catty Version That Doesn't Work - V1

Here is a natural way to think of the code, but it does not work:

```
def catty(s):  
    result = ''  
    for i in range(len(s)):  
        if s[i] == 'c' or s[i] == 'a' or s[i]  
        == 't':  
            result += s[i]
```

```
return result
```

What is the problem? Upper vs. lower case. We are not getting any uppercase chars 'C' for example.

Catty Solution V2

Solution: convert each char to lowercase form `.lower()`, then test.

Solution - this works, but that if-test is quite long, can we do better? Indeed, it's so long, it's awkward to fit on screen.

Aside: see style guide [breaking up long lines](#) for way to break up long lines like this.

```
def catty(s):
    result = ''
    for i in range(len(s)):
        if s[i].lower() == 'c' or
s[i].lower() == 'a' or s[i].lower() == 't':
            result += s[i]
    return result
```

Idea: Decomp By Var

- The code is getting a little lengthy
- The repeated `s[i].lower()` is irksome
- Compute the value once, store in a variable
- Advantages
 - 1. Shorten the code, less repetitive typing
 - 2. Variable name helps the code "read" better
 - 3. A sort of decomp within a function - break the big thing into little steps

Decomp Var Steps

- Some phrase X repeated in code
e.g. `s[i].lower()`
- Create a variable, compute phrase once and store
`low = s[i].lower()`
- Use that variable on later lines
- Variable name noun - code is more readable

Catty Solution V3 - Better

Start with the v2 code. Create variable to hold the repeated computation - shorten the code and it "reads" better with the new variable.

```
low = s[i].lower()
```

The complete solution

```
def catty(s):  
    result = ''  
    for i in range(len(s)):  
        low = s[i].lower()    # decomp by var  
        if low == 'c' or low == 'a' or low ==  
't':  
            result += s[i]  
    return result
```

We will use this strategy frequently with CS106A code. If the solution is getting a little lengthy, introduce a variable for some sub-part of the computation like this.

Style: Variable Names

The name of a variable should label that value in the code, helping the programmer to keep their ideas straight. Other than that, the name can be short. The name does

not need to repeat every true thing about the value. Just enough to distinguish it from other values in this algorithm.

1. Good names for this example, short but with key facts: **low**, **low_char**
2. Names with more detail, probably too long: **low_char_i**, **low_char_in_s**
3. Avoid this name: **lower** - name is ok, but avoid using a name which is the same as the name of a function, just to avoid confusion.

The V2 code above is acceptable, but V3 is shorter and nicer. The V3 code also runs slightly faster, as it does not needlessly re-compute the lowercase form three times per char.

Optional Aside: "in" Trick Form of "or"

This is just coding trick, not something we would ever require or look for students to do. The **in** can be do the "or" logic for us, like this:

```
# instead of this
if low == 'c' or low == 'a' or low == 't':
    ...

# this works - a trick use of "in"
if low in 'cat':
    ...
```

Recall: if and if/else:

- Have the plain `if` is the most common - one test and one action
- Then have `if/else:` - one test selecting between two actions
- Those are the most common
- Here is a third `if/elif` structure in case there are N tests to go through

N Tests - if/elif

- N tests to check - `if/elif` structure
- This situation is not that common, but if you have N tests, this is the structure
- Go through N tests
Try each test until one is `True`
- Evaluate `test1`, `test2`, `test3`
- As soon as a test is `True`
Run that action
Exit the `if/elif` structure
No further tests/actions will run
- Optional `"else:"` at end runs if no test is true
- Mnemonic: `"else"` and `"elif"` are the same length

```
if test1:
    action1
elif test2:
    action2
elif test3:
    action3
else:
    action4
```

```
# Tries test1, if that's False, tries test2,
# if that's False, tries test3, and so on.
```

Example: `vowel_swap()`

> [`vowel_swap\(\)`](#)

The most common letters used in English text are: **e, t, a, i, o, n**

Here we process string s, swapping around the 3 vowels like this:

```
e -> a
a -> i
i -> e
```

This changes an English word in a way that looks like a word and is kind of funny.

```
'kitten' -> 'kettan'
'table' -> 'tibla'
'radio' -> 'rideo'
```

`vowel_swap(s)`: Given string `s`. We'll swap around the three most common vowels in English, which are 'e', 'a', and 'i'. Return a form of `s` where each lowercase 'e' is changed to 'a', each 'a' is changed to 'i', and each 'i' is changed to 'e'. Other chars leave unchanged. So the word 'kitten' returns 'kettan'. Use an if/elif structure. The basic string loop is provided.

vowel_swap() Solution

```
def vowel_swap(s):
    result = ''
    for i in range(len(s)):
        if s[i] == 'e':
            result += 'a'
        elif s[i] == 'a':
            result += 'i'
        elif s[i] == 'i':
            result += 'e'
        else:
            result += s[i]
    return result
```

if/elif vs. if/return

The use of `return` can accomplish something similar to the if/elif structure, which is why we have not really needed if/elif. Suppose we are doing the vowel-swap

algorithm, but in a function that processes a single char. This code works perfectly without using if/elif:

```
def swap_ch(ch):  
    """Vowel-swap on one char."""  
    if ch == 'e':  
        return 'a'  
    if ch == 'a':  
        return 'i'  
    if ch == 'i':  
        return 'e'  
    return ch
```

Since the return exits the function - we get the if/elif feature that once a test is taken, the later tests are skipped.

However, the full-string `vowel_swap()` above cannot use `return` like this, as it needs to keep running the loop to do the other characters. We need to handle each char in the loop but without leaving the function, and for that, the if/elif is perfect.

Later Practice: `str_adx()`

> [`str_adx\(\)`](#).

`str_adx(s)`: Given string `s`. Return a string of the same length. For every alphabetic char in `s`, the result has an 'a', for every digit a 'd', and for every other type of char the result has an 'x'. So 'Hi4!x3' returns 'aadxad'. Use an if/elif structure.

- If/elif logic to check for different char types
- alpha char → 'a', digit → 'd', otherwise → 'x'
- e.g. 'Z5\$\$t' → 'adxxa'

Solution

```
def str_adx(s):  
    result = ''
```

```
for i in range(len(s)):
    if s[i].isalpha():
        result += 'a'
    elif s[i].isdigit():
        result += 'd'
    else:
        result += 'x'
return result
```

String .find()

'Python' (len 6)

P	y	t	h	o	n
0	1	2	3	4	5

- `s.find(target_str)` - search `s` for `target_str`
- Returns int index where found first, searching from start of `s`
- Returns -1 if not found
- Alternate form: 2nd "start_index" parameter, starts search from there
`s.find(target_str, start_index)` (show on later example)

```
>>> s = 'Python'
>>>
>>> s.find('th')
2
>>> s.find('o')
4
>>> s.find('y')
```

```
1
>>> s.find('x')
-1
>>> s.find('N')
-1
>>> s.find('P')
0
```

Strategy: Dense = Slow Down

- Some lines of code are routine
- Require just normal attention
- An advantage of using idiomatic phrases
`for i in range(len(s)):`
- **But** some lines are dense
- Slow down for those, work carefully
- Slices (below) are dense!
- Dense = Powerful!

Python String Slices

- This is a fantastic feature
- "substring" - contiguous sub-part of a string
- Access substring with 2 numbers
- "slice" uses colon to indicate a range of indexes
- `s[1:3]` returns `'yt'`
- **Start at first number**
- **Up to but not including second number** UBNI

- `s[3:3]` = empty string
"Not including" dominates the "starting at"
- Try it in the interpreter
- Style: typically written with no spaces around ":"

`'Python' (len 6)`

P	y	t	h	o	n
0	1	2	3	4	5

```
>>> s = 'Python'
>>> s[1:3]      # 1 .. UBNI
'yt'
>>> s[1:5]
'ytho'
>>> s[4:5]
'o'
>>> s[4:4]      # "not including" dominates
''
```

Slice Without Start/End

- If start index is omitted, goes from start of string
- If end index is omitted, goes through end of string
- If number is too big .. uses end also
This is a little unusual
In other cases, Python will halt with an error if an index is too big, but not with slices
- Note perfect split: `s[:4]` and `s[4:]`
Number used as both start and end
Splits the string into 2 pieces exactly

'Python' (len 6)

P	y	t	h	o	n
0	1	2	3	4	5

```
>>> s[:3]          # omit = from/to end
'Pyt'
>>> s[4:]
'on'
>>> s[4:999]       # too big = through the end
'on'
>>> s[:4]          # "perfect split" on 4
'Pyth'
>>> s[4:]
'on'
>>> s[:]           # the whole thing
'Python'
```

Example: brackets()

A first venture into using index numbers and slices. Many problems work in this domain - e.g. extracting all the hashtags from your text messages.

'cat[dog]bird' -> 'dog'

> [brackets](#)

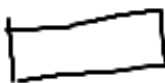
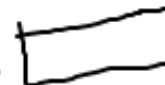
brackets(s): Look for a pair of brackets '['...']' within s, and return the text between the brackets, so the string 'cat[dog]bird' returns 'dog'. If there are no brackets, return the empty string. If the brackets are present, there will be only one of each, and the right bracket will come after the left bracket.

- Problem spec: either 2 brackets, or zero brackets

- Drawing of 'cat[dog]bird' to work out steps
- Strategy:
- Use s.find()

```
left = s.find('[')
right = s.find(']')
```
- Switch between drawing and code
- Decomp by var
 Store in variable `left` for later lines
 Nice to have words `left` and `right` in code narrative
- Look for right bracket
- Use slice to pull out and return answer

Brackets Drawing

`left` 
`right` 

`'cat[dog]bird'`
 0 1 2 3 4 5 6 7 8 9

Brackets Observations

- Make a drawing - work out the index numbers
- Drawing + code strategy

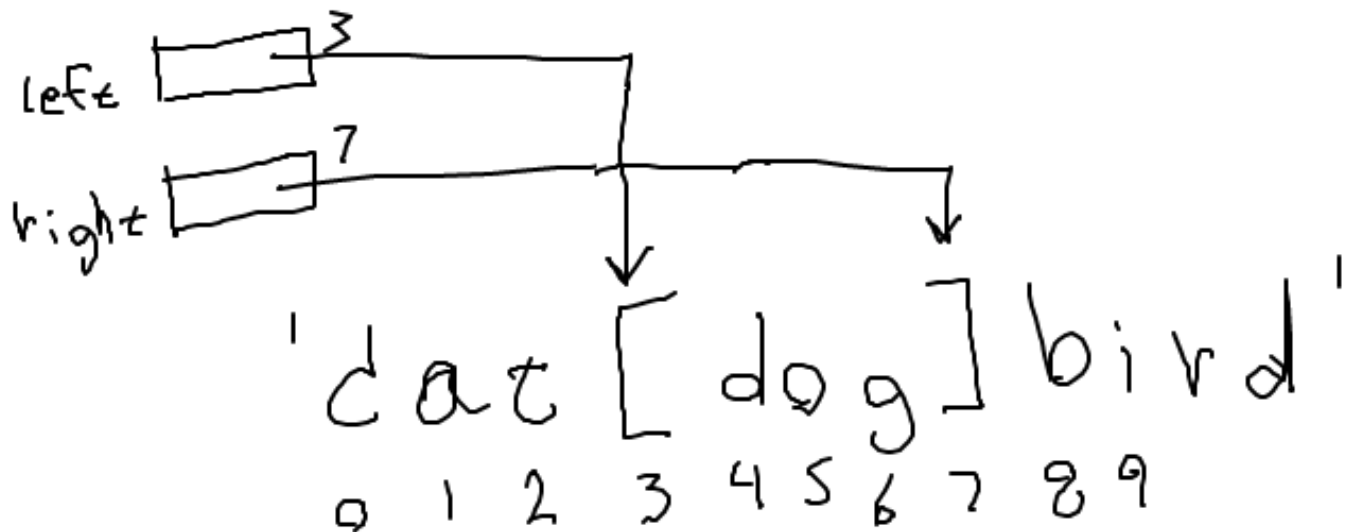
- Code should work in general
- BUT can use specific string to work out numbers
- e.g. 'cat[dog]bird'
- Empty string input - works?
- What about input 'a[]z'
Verify that our slice works here too

Brackets Solution + Readable

```
def brackets(s):  
    left = s.find('[')  
    if left == -1:  
        return ''  
    right = s.find(']')  
    return s[left + 1:right]
```

We prefer "readable" code — when the eye sweeps over the code, what the code does is apparent. In this case, the named variables help make the code readable. Look at that last line, how the variables help spell out how the slice pulls in the data from the previous lines. Code that is readable as you write it contains fewer bugs.

Brackets Drawing After



Brackets Without Decomp-by-var

The variables `left` and `right` are a very natural example of decomp-by-var. In the code, the variables name an intermediate value that runs through the computation.

Below is what the code looks like without the variables. It works fine, and it's one line shorter, but the readability is clearly worse. It also likely runs a little slower, as it computes the left-bracket index twice.

```
def brackets(s):  
    if s.find('[') == -1:  
        return ''  
    return s[s.find('[') + 1: s.find(')']]
```

The first solution with its variables looks better and is more readable.

Aside: Off By One Error

We have seen many examples of int indexing to access a part of a structure. So of course doing it slightly wrong is very common as well. So common, there is a phrase for it - "off by one error" or OBO — it even has its own [wikipedia page](#). You

can feel some kinship with other programmers each time you stumble on one of these.

"My code is perfect! Why is this not working? Why is this not work ... oh, off by one error. We meet again!"

Why do you need to think of new ways for your code to go wrong when the old ways work so well!

Exercise: at_3()

> [at_3](#)

Here is a problem similar to brackets for you to try. If we have enough time in lecture, we'll do it in lecture. A drawing really helps the OBO on this one.

Milestone-1 - get the 'abc' output below, not worrying about if the input is too short

Milestone-2 - add logic for the too-short case. Note the `i < len(s)` valid idea below.

`at_3(s)`: Given string `s`. Find the first '@' within `s`. Return the len-3 substring immediately following the '@'. Except, if there is no '@' or there are not 3 chars after the '@', return ''.

```
'xx@abcd' -> 'abc'
'xxabcd'   -> ''
'x@x'      -> ''
```

Valid Index: `i < len(s)`

- Zero based indexing, say for a string
- Valid index numbers are 0, 1, 2, ... len-1
- This means for a non-negative `i`:
`i < len` tests if `i` is valid
Often see `<` in tests for this "valid" pattern

More `s.find()` if we have time...

`s.find()` 2 Param Form

`s.find()` variant with 2 params: `s.find(target, start_index)` - start search at `start_index` vs. starting search at index 0. Returns -1 if not found, as usual. Use to search in the string starting at a particular index.

Suppose we have the string `'[xyz['`. How to find the second `'['` which is at 4? Start the search at 1, just after the first bracket:

```
>>> s = '[xyz['
>>> s.find('[')           # find first [
0
>>> s.find('[', 1)       # start search at 1
4
```

Exercise: parens()

> [`parens\(\)`](#).

`'x)x(abc)xxx' -> 'abc'`

This is nice, realistic string problem with a little logic in it.

Thinking about this input: `'))(abc)'`. Starting hint code, something like this, to find the right paren after the left paren:

```
left = s.find('(')
...
right = s.find(')', left + 1)
```

Optional: Negative Slice

P	y	t	h	o	n
0	1	2	3	4	5
(-6	-5	-4	-3	-2	-1)

- Optional / advanced shorthand - you never need to use this
- Handy way to refer to chars near **end** of string
- Negative numbers to refer to chars at end of string
- -1 is the last char
- -2 is the next to last char
- Works in slices etc.
- Maybe just memorize this one:
s[-1] is the last char in s

```
>>> s = 'Python'
>>> s[len(s) - 1]
'n'
>>> s[-1] # -1 is the last char
'n'
>>> s[-2]
'o'
>>> s[-3]
'h'
>>> s[1:-3] # works in slices too
'yt'
>>> s[-3:]
'hon'
```