UNIVERSITY OF WATERLOO

Lesson 3.1: Strings — Definition and Operations

Introduction to Strings

In this module, we will go into greater depth with strings. We recall the definition from Module 1 and we use this definition to help us solve problems involving strings using recursion. We then discuss some of the powerful tools Python has built in to help process strings. In the subsequent two lessons of this module, we get our first glimpse into side effects in Python via `print` and `input`.

Concept Check 3.1.1

1/1 point (graded)
Which of the following gives our formal definition of Python strings?

- ○ A Python string is anything consisting of quotes and characters.
- ○ A Python string is any text that can be processed with a computer.
- ○ A Python string is a piece of fabric used to make spools.
- ○ A Python string is any of the characters found on a common keyboard joined together to form an object.
- ◉ A Python string is either the empty string or a single character added to another string. ✔

✔ Correct (1/1 point)

Strings in Python: Combining strings in interesting ways

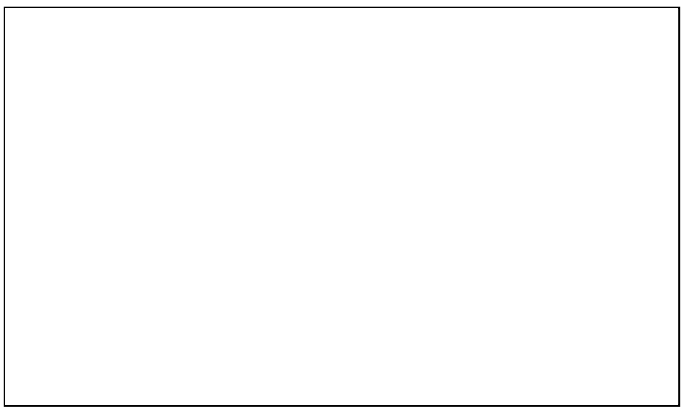
Note that strings can be defined using either single or double quotes (but must start and end with the same kind of quote). Python might interchange between the two but know that a string is enclosed by one set of quotes (and they must be the same quote).

Notice below that you can add strings. Can you subtract strings? Would this be reasonsable for any two arbitrary strings?

```
1  s = "Great"
2  t = "CS116"
3  u = s + t
4  v = s + "!!!! " + t
5  x = 'single quote works too'
6  y = 'strings can contain quotes" too'
7  z = "strings can contain even contain single
```

<< First Step | < Step Back | Done
Step Forward > | Last Step >> | Close Visualizer

Code Output



Global frame

s	"Great"
t	"CS116"
u	"GreatCS116"
v	"Great!!!! CS116"
x	"single quote works too"
y	"strings can contain quotes\"
z	"strings can contain even con

Overloading Operators

In Python, as the above shows, an operation like + can be used in multiple contexts. In this way, you can think of an operation like + as having multiple valid contracts, extending our traditional definition of contract for functions.

Concept Check 3.1.2

1 point possible (graded)
Which of the following are valid contracts for `+` ? **Check all that apply**.

- [ ] `+: Int Int -> Int`
- [ ] `+: Float Float -> Int`
- [ ] `+: Int Float -> Float`
- [ ] `+: Str Int -> Str`
- [ ] `+: Str Str -> Str`

Concept Check 3.1.3

1 point possible (graded)
Which of the following are valid contracts for `-` ? **Check all that apply**.

- [ ] `-: Int Int -> Int`
- [ ] `-: Float Float -> Int`
- [ ] `-: Int Float -> Float`
- [ ] `-: Str Int -> Str`
- [ ] `-: Str Str -> Str`

In what preceeded this, we can see that multiple contracts are valid for + and -. In computer science, we say that + and - are **overloaded operators**. We have seen another operator that was overloaded and that is the * operator.

Concept Check 3.1.4

1 point possible (graded)
Which of the following are valid contracts for `*`? **Check all that apply**.

- [ ] `*: Int Int -> Int`
- [ ] `*: Float Float -> Int`
- [ ] `*: Int Float -> Float`
- [ ] `*: Float Int -> Float`
- [ ] `*: Float Float -> Float`

Concept Check 3.1.5

1 point possible (graded)
It turns out that `*: Int Str -> Str` and `*: Str Int -> Str` are also valid contracts for `*` as well! What do you think the most reasonable answer for `"happy" * 2` is?

- ( ) `"ha"`
- ( ) `"pp"`
- ( ) `"happy"`
- ( ) `"happyhappy"`
- ( ) `"happyhappyhappy"`

```
1  s = "repeat"
2  t = s * 3
3  u = 3 * s
4  w = 0 * "test"
5  x = "negatives?" * (-1)
```

Visualize Reset Code

Concept Check 3.1.6

1 point possible (graded)
Based on the above, what would the value of `"happy" * (-2)` be?

- ( ) `""`

- `"ha"`
- `"ah"`
- `"happyhappy"`
- `"yppahyppah"`

There are lots of questions we might want to ask about strings and Python usually gives us the tools to answer these questions quickly. A common question is to ask whether or not a string is a substring of another string. A substring is a sequence of contiguous characters inside a string. Checking if a string `s` is a substring of a string `t` can be done using the `in` operation. This operation is a relational operator (like `<, ==, >` etc.).

Examples:

```
"astro" in "catastrophe" => True
"car" in "catastrophe" => False
 "" in "catastrophe" => True
```

Another operation that we saw in Module 1 was the `len` function, which returns the number of characters in a given string.

Examples:

```
len("catastrophe") => 11
len("cat astro phe") => 13
len("") => 0
```

Concept Check 3.1.7

1 point possible (graded)
What is the value of `("rest" in "interesting") and ("test" in "interesting")`?

- `True`
- `False`
- There is an error with the code above.
- Cannot be determined

Concept Check 3.1.8

1 point possible (graded)
What is the value of `"rest" in "interesting" == True`?

- `True`
- `False`
- There is an error with the code above.

- ◯ Cannot be determined

Concept Check 3.1.9

1 point possible (graded)
What is the value of `("rest" in "interesting") == True`?

- ◯ `True`
- ◯ `False`
- ◯ There is an error with the code above.
- ◯ Cannot be determined

Concept Check 3.1.10

1 point possible (graded)
Which of the following expressions evaluates to `True` if and only if the length 1 string `t` is one of the first 4 letters in the alphabet? **Check all that apply**.

- ☐ `t == 'a' or 'b' or 'c' or 'd'`
- ☐ `t == 'a' or t == 'b' or t == 'c' or t == 'd'`
- ☐ `t in 'abcd' == True`
- ☐ `t in 'abcd'`

String Comparisons

In Python, we can also compare strings using <, <=, ==, >, >=, and !=. These will make comparisons in lexicographic order (dictionary order) based on the following additional rules:

- The numbers 0-9 come first
- Capital letters in order from A-Z come next
- Lower case letters follow last
- Lastly, if there are ties, compare character by character until one string appears first in the dictionary. Note shorter strings come before longer strings (for example `"tie" < "ties"`)

For more about why this order is, please look up an ASCII table (We will discuss this more in Module 10). As an example, the following are all true:

```
"0" < "a"
"Zebra" < "apple"
"Apple" < "Banana"
"apple" < "apples"
"banana" < "zebra"
```

Concept Check 3.1.11

1 point possible (graded)
What is the value of `"24601" <= "Jean Valjean"`?

- `True`
- `False`
- There is an error with the code above.
- Cannot be determined

Concept Check 3.1.12

1 point possible (graded)
What is the value of `"CS116" >= "CS116"`?

- `True`
- `False`
- There is an error with the code above.
- Cannot be determined

Concept Check 3.1.13

1 point possible (graded)
What is the value of `"Banana" != "banana"`?

- `True`
- `False`
- There is an error with the code above.
- Cannot be determined

A question you might ask is whether or not it is possible to access individual characters in the string. The answer is yes and we do this using a square bracket notation. In Python, string positions are labelled by both positive and negative indices. Indices in Python begin at 0, the smallest natural number and end with `len(s) - 1`, where `s` is the string we are considering. We can also index characters negatively where the index -1 is the last character and `-len(s)` is the first character. The following is an explicit example of which letter occurs at which index for the string `'four'`

index	0	1	2	3
string	`f`	`o`	`u`	`r`

index	-4	-3	-2	-1

In Python notation, if `s = "four"` then

- `s[0] == s[-4] == 'f'`
- `s[1] == s[-3] == 'o'`
- `s[2] == s[-2] == 'u'`
- `s[3] == s[-1] == 'r'`

Play around with the code below to see some of the different values of string indexing. What happens when you access a string position that is outside the valid range?

```
1  s = "unpredictably"
2  t = s[0]             # t == 'u'
3  u = s[len(s)-1]      # u == 'y'
4  v = s[-1]            # v == 'y'
5  w = s[-len(s)]       # w == 'u'
6  x = s[10]            # x == 'b'
7  y = s[len(s)]        # Error
8  z = s[-len(s)-1]     # Error
9  a = ""               # a == ''
10 b = a[0]             # Error
```

Visualize Reset Code

Concept Check 3.1.14

1 point possible (graded)
What is the full range of valid indices for the square bracket index `s[i]` of a string `s` for integers `i`?

- `0 <= i < len(s)`
- `1 <= i < len(s)`
- `0 <= i <= len(s)`
- `-len(s) <= i < len(s)`
- `-len(s) <= i <= len(s)`

Concept Check 3.1.15

1 point possible (graded)
What character corresponds to `"play"[2]`?

- `"p"`
- `"l"`

- ◯ `"a"`
- ◯ `"y"`
- ◯ Error

Concept Check 3.1.16

1 point possible (graded)
What character corresponds to `""[0]`?

- ◯ `""`
- ◯ `''`
- ◯ `"0"`
- ◯ Error
- ◯ More than one of the above

Slicing Strings

Python also gives us a handy way to extract substrings from a given string `s` using slicing notation. Like with indexing, positions can be non-negative or negative and on the non-negative side beginning with 0.

The notation `s[i:j]` returns the substring from string `s`, containing all the characters in positions `i, i+1, i+2, ..., j-1` . In particular, notice that the last index is not a part of the substring returned from the slicing notation.

- If the starting index is not given, Python will return the substring that begins with 0 and ends with one character before the end index.
- If the end index is not given, Python will begin with the start index and end with the end of the string.
- If both the start and end index are not given, a copy of the string is returned.
- If the start index appears after the end index in a string, then the empty string is returned. Notice that this can cause confusion if dealing with both positive and negative indices so be careful!
- If indices are out of range, Python will take all characters that exist in the range. If none exist, the empty string is returned.

```
1  s = "unpredictably"
2  t = s[0:3]            # t == 'unp'
3  u = s[3:]             # u == 'redictably'
4  v = s[3:len(s)-1]     # v == 'redictabl'
5  w = s[3:9999]         # w == 'redictably'
6  x = s[:-1]            # x == 'unpredictabl'
7  y = s[:]              # y == 'unpredictably'
8  z = s[:0]             # z == ''
9  a = s[-10:10]         # a == "redicta"
10 b = s[-1000:1000]     # b == 'unpredictably'
```

```
11  c = s[1000:1010]        # c == ''
```

Visualize | Reset Code

Concept Check 3.1.17

1 point possible (graded)
What is the full range of valid indices for the slicing `s[i:j]` of a string `s` for integers `i` and `j`?

- `0 <= i, j < len(s)`
- `0 <= i, j <= len(s)`
- `-len(s) <= i, j < len(s)`
- `-len(s) <= i, j <= len(s)`
- All integers `i` and `j` will work

Concept Check 3.1.18

1 point possible (graded)
What is the value of `"play"[:]`?

- `""`
- `"p"`
- `"pla"`
- Error
- None of the above

Concept Check 3.1.19

1 point possible (graded)
What is the value of `"play"[1:]`?

- `""`
- `"p"`
- `"lay"`
- Error
- None of the above

Concept Check 3.1.20

1 point possible (graded)
What is the value of `"play"[-2:]`?

- ◯ `"ay"`
- ◯ `"pl"`
- ◯ `ya"`
- ◯ Error
- ◯ None of the above

Concept Check 3.1.21

1 point possible (graded)
What is the value of `"play"[-7:-5]`?

- ◯ `"ay"`
- ◯ `"pl"`
- ◯ `"ya"`
- ◯ Error
- ◯ None of the above

Concept Check 3.1.22

1 point possible (graded)
What is the value of `"play"[:1]`?

- ◯ `""`
- ◯ `"p"`
- ◯ `"y"`
- ◯ Error
- ◯ None of the above

Concept Check 3.1.23

1 point possible (graded)

What is the value of `""[:1]`?

- `""`
- `"p"`
- `"y"`
- Error
- None of the above

Concept Check 3.1.24

1 point possible (graded)
What is the value of `""[0:1]`?

- `""`
- `"p"`
- `"y"`
- Error
- None of the above

Lastly, Python gives us a way to take subsequences of characters from a string using the slicing notation with a step.

The notation `s[i:j:k]` returns the subsequence of characters from the string `s`, containing all the characters in positions from `i` up to but not including `j` and we count by `k` characters. In particular, notice that the last index is not a part of the substring returned from the slicing notation.

- If the stepping index is not given, Python will count by one.
- If the stepping index is negative, then you perform the skipping from right to left instead of from left to right. That is, reverse the roles of `i` and `j` above. Note that this also means that the start index should be more to the right of the string than to the left.
- Notice that `s[::-1]` will return the string with the characters in reversed order.

This can get complicated when mixing positive and negative values for the values of `i`, `j` and `k`. Some of the concept checks below explore this and are tricky. Try them but don't fret about these unusual cases. Focus on understanding the most common uses of slicing. That is, when `0 <= i <= j <= len(s)` and `k > 0` are `True`.

```
1  s = "troublemaking"
2  t = s[0:10:2]          # t == 'tobea'
3  u = s[0:10:10]         # u == 't'
4  v = s[0:10:9]          # v == 'tk'
5  w = s[::-1]            # w == 'gnikamelbuort'
6  x = s[-10:10:3]        # x == 'uek'
```

```
7  y = s[2:6:2]          # y == 'ob'
8  z = s[2:6:-2]         # z == ''
9  a = s[6:2:-2]         # a == 'eb'
10 b = s[-4:-9:-1]       # b == 'kamel'
11 c = s[-9:-4:-1]       # c == ''
```

Visualize | Reset Code

Concept Check 3.1.25

1 point possible (graded)
What is the value of `"flowcharting"[::3]`?

- `"ohtg"`
- `"lcrn"`
- `"fwai"`
- Error
- None of the above

Concept Check 3.1.26

1 point possible (graded)
What is the value of `"flowcharting"[::-1]`?

- `"flowcharting"`
- `"gnitrahcwolf"`
- `"girhwl"`
- Error
- None of the above

Concept Check 3.1.27

1 point possible (graded)
What is the value of `"flowcharting"[2:5:-1]`?

- `""`
- `"cwo"`
- `"hcw"`
- `"owc"`

- ◯ Error

Concept Check 3.1.28

1 point possible (graded)
What is the value of `"flowcharting"[-3:5:-1]`?

- ◯ `"itra"`
- ◯ `"arti"`
- ◯ `"itrah"`
- ◯ Error
- ◯ None of the above

Concept Check 3.1.29

1 point possible (graded)
What is the value of `'e' == 'a' or 'e' or 'i' or 'o' or 'u'`?

- ◯ `True`
- ◯ `False`
- ◯ Error
- ◯ Cannot be determined
- ◯ None of the above

Immutability

In Python, there are mutable and immutable data types. Immutable ones are those we cannot change (as an example, it would be strange to change the meaning of say the number 1 and so the data type `Int` is immutable. Notice that `Int, Float, Str,` and `Bool` values are all immutable.

You might want to write something like:

```
1  s = "play"
2  s[0] = "c"
```

In hopes to change the word `"play"` to `"clay"` however this will give you an error that strings do not support item assignment (reworded, strings are immutable).

What one can do is the following:

```
1  s = "play"
2  s = "c" + s[1:]
```

This will produce the new string `"clay"` and store it in `s`

In later lessons, we will learn about other data types like lists, that are mutable. These will vastly complicate our memory model diagrams used in the visualizer tool.

Concept Check 3.1.30

Write the body of a function `pig_latin(s)`, which returns the Pig Latin version of the English word `s`. You may assume that `s` consists of lower case Latin alphabet letters and is nonempty (that is, of size at least 1). For the purposes of this question, our version of Pig Latin consists of the following two rules:

- If the word begins with a consonant (including `'y'`), the consonant is removed from the beginning and placed at the end of the word and the letters `'ay'` are added.
- If the word begins with a vowel (excluding `'y'`), simply add the letters `'way'` to the end of the word.

0/3 points
Attempts: 0 / Unlimited

```
1  def pig_latin(s):
2    '''
3    Given a lower case Latin alphabet string s, return the
4    Pig Latin equivalent of the word.
5
6    pig_latin: Str -> Str
7    Requires:
8       s contains only letters from the lower case Latin alphabet.
9       s is nonempty.
10
11   Examples:
12      pig_latin('apple') => 'appleway'
13      pig_latin('your') => 'ouryay'
14      pig_latin('sun') => 'unsay'
15   '''
16   ##YOUR CODE GOES HERE
17   pass
```

Concept Check 3.1.31

The function `swap_two(s, pos1, pos2)` consumes a string `s` of length at least 2 and two positions `pos1` and `pos2`, two natural numbers between 0 and `len(s)-1` inclusive, and returns a string with the character in those two positions exchanged.

```
1  def swap_two(s, pos1, pos2):
2    '''
3    Returns s with s[pos1] and s[pos2] swapped.
4
5    swap_two: Str Nat Nat -> Str
6    Requires:
7       len(s) >= 2
8       0 <= pos1 <= len(s) - 1
```

```
9       0 <= pos2 <= len(s) - 1
10
11    Examples:
12       swap_two("ab", 0, 1) => "ba"
13       swap_two("banana", 2, 5) => "baaann"
14    '''
15    ##BODY OF FUNCTION IS HIDDEN
```

Write a complete black-box testing suite for `swap_two` using the `check` module.

Remember that the number of tests you include is not important. To be marked correct, your test suite has to be of sufficient quality. We measure this by identifying specific tests that any proper testing suite of `swap_two` should include. In particular, you may want to revisit Lesson 1.4 and its discussion of typical values, small values, large values, boundary values, special cases and relationships between values.

All of your tests must agree with the contract and requirements of `swap_two`. Lastly, make sure you use your testing suite for the next problem!

0/3 points
Attempts: 0 / Unlimited

```
1  #use but do not import the check module
```

Concept Check 3.1.32

Write the body of a function `swap_two(s, pos1, pos2)` that consumes a string `s` of length at least 2 and two positions `pos1` and `pos2`, two natural numbers between 0 and `len(s)-1` inclusive, and returns a string with the character in those two positions exchanged.

Hint: Do not make any assumptions about the positions that aren't mentioned in the requirements! Test your code with **all** possible scenarios for `pos1` and `pos2`.

0/3 points
Attempts: 0 / Unlimited

```
def swap_two(s, pos1, pos2):
    '''
    Returns s with s[pos1] and s[pos2] swapped.

    swap_two: Str Nat Nat -> Str
    Requires:
       len(s) >= 2
       0 <= pos1 <= len(s) - 1
       0 <= pos2 <= len(s) - 1

    Examples:
       swap_two("ab", 0, 1) => "ba"
       swap_two("banana", 2, 5) => "baaann"
    '''
    ##YOUR CODE GOES HERE
    pass
```

Concept Check 3.1.33

1 point possible (graded)
Consider the following code which contains several errors:

```
def lots_of_errors(s):
  s = s[10]
error = "banana" + lots_of_errors("banana") + "banana"[:10] + foo()
```

Python reports errors as it first encounters them. Based on this, what error would be displayed if we ran the above code?

- ◯ *TypeError: Can't convert 'NoneType' object to str implicitly (or unsupported operand type(s) for +: 'str' and 'NoneType')*
- ◯ *IndexError: string index out of range*
- ◯ *IndexError: string slice index out of range*
- ◯ *NameError: name 'foo' is not defined*

Concept Check 3.1.34

1 point possible (graded)
Consider the following code:

```
answer = (123[1])
```

What is the value of `answer` if we run the code above?

- ◯ `1`

- ◯ `2`
- ◯ `3`
- ◯ We get the error *TypeError: 'int' object is not subscriptable*.
- ◯ None of the above

In this lesson, we discussed many operations that can be done with strings, including slicing which helps us to get substrings and subsequences of characters from given strings.

In the next two lessons, we will discuss our first commands that produce side effects, namely `print` and `input`. In the final lesson, we will talk about even more string methods that Python has to help process strings and finally solve problems with strings using structural recursion via the definition of a string.