Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 3 Study Guide and Deliverables

Theme:	Building Python Projects
Readings:	• Chapter 2 (pp. 122-140), Chapter 4, Chapter 6 (pp. 271-276), Chapter 14 (pp. 645-665) • Module Lecture Notes
Topics:	Strings, Collections, Control Flow, Iterations, Files, Lists
Assignments	• Assignment 3 due on Tuesday, April 6 at 6:00 PM ET • Final Project Topic due on Wednesday, April 7 at 6:00 PM ET
Assessments	Quiz 3: • Available Friday, April 2 at 6:00 AM ET • Due on Tuesday, April 6 at 6:00 PM ET
Live Classrooms:	• Tuesday, March 30, 8:00 - 9:30 PM ET • Thursday, April 1, 6:00 - 7:30 PM ET • Facilitator Session: Friday, April 2, at 8:00 PM ET

Learning Objectives

After successfully completing this module, the learner is expected to do the following:

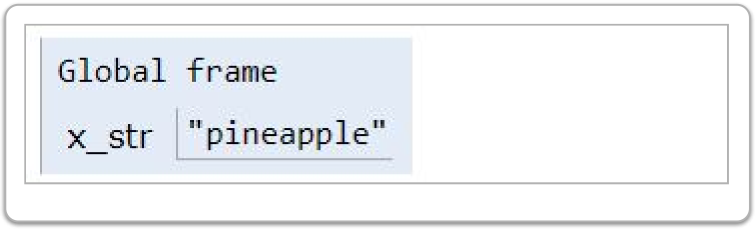
- Manipulate text with strings and string functions.
- Use collections in programing.
- Use control flow.
- Perform iterations.
- Open, access, read, and store files.
- Apply list in different applications.

Strings and Text Manipulation

Strings - Python String Overview

- A Python string is an object, not just an array of character data.
- A string is ordered and immutable.
- There are many built-in methods in Python to manipulate strings.

```
x_str = 'pineapple'
```



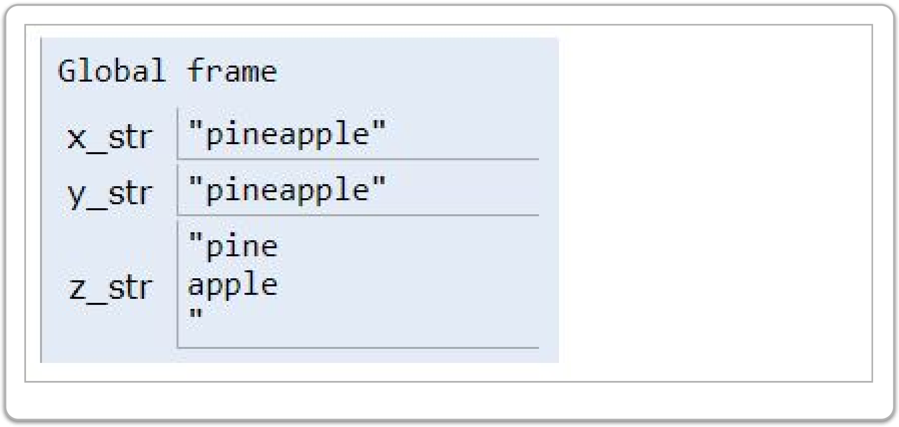
Global frame

x_str "pineapple"

0	1	2	3	4	5	6	7	8
p	i	n	e	a	p	p	l	e

Defining Strings

```
x_str = 'pineapple' # single quote
y_str = "pineapple" # double quote
# triple quotes allow multi-line strings
z_str = """ pine
apple
"""
```



```
Global frame
x_str  "pineapple"
y_str  "pineapple"
       "pine
z_str  apple
       "
```

Test Yourself Exercises

Test Yourself 3.1.01

Write programs to show three ways to define the following (old English proverb) string *x_str*:

> "after
>
> meat
>
> comes
>
> mustard"

Suggested program:

First way:

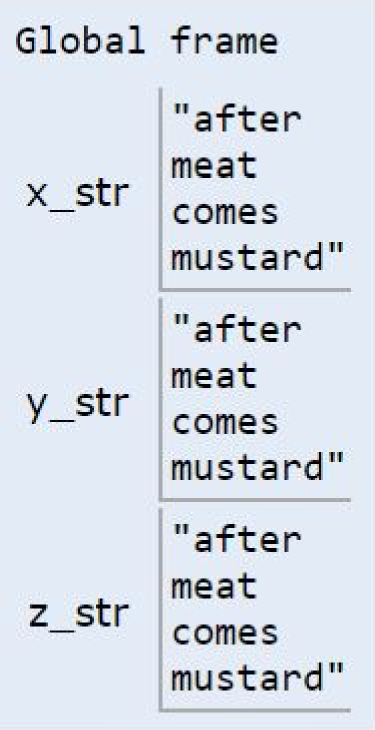
```
x_str =""" after
meat
comes
mustard """
```

Second way:

```
y_str = "after" + "\n" + "meat" + "\n" + \
           "comes" + "\n" + "mustard"
```

Third way:

```
z_str =""" after
meat """ + """
comes
mustard """
```



Global frame

x_str	"after meat comes mustard"
y_str	"after meat comes mustard"
z_str	"after meat comes mustard"

Test Yourself 3.1.02

How may newline characters are there in *x_str*?

Suggested answer: three newline characters.

String Encoding

Every character is mapped to an integer.

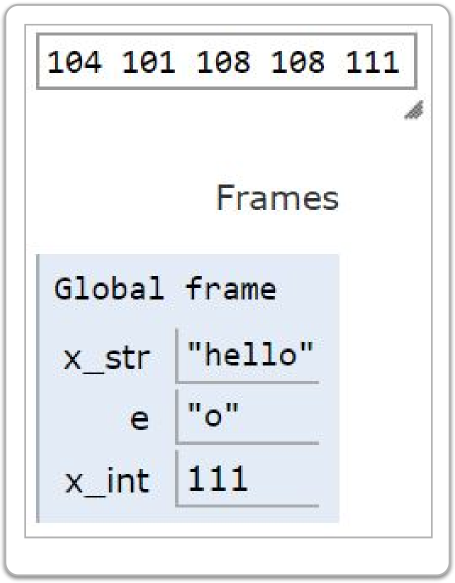
- Past: ASCII code for each charcater
- Now: UTF variable length encoding
    a. international alphabets
    b. memory efficiency

ord() and *chr()* are used for forward and reverse mapping.

ord() Function

ord(): maps character to its integer "value".

```
# print integer values
# for each character
x_str = 'hello'
for e in x_str:
    x_int = ord(e)
    print(x_int, end = " ")
```



```
104 101 108 108 111
```

Frames

Global frame

x_str "hello"

e "o"

x_int 111

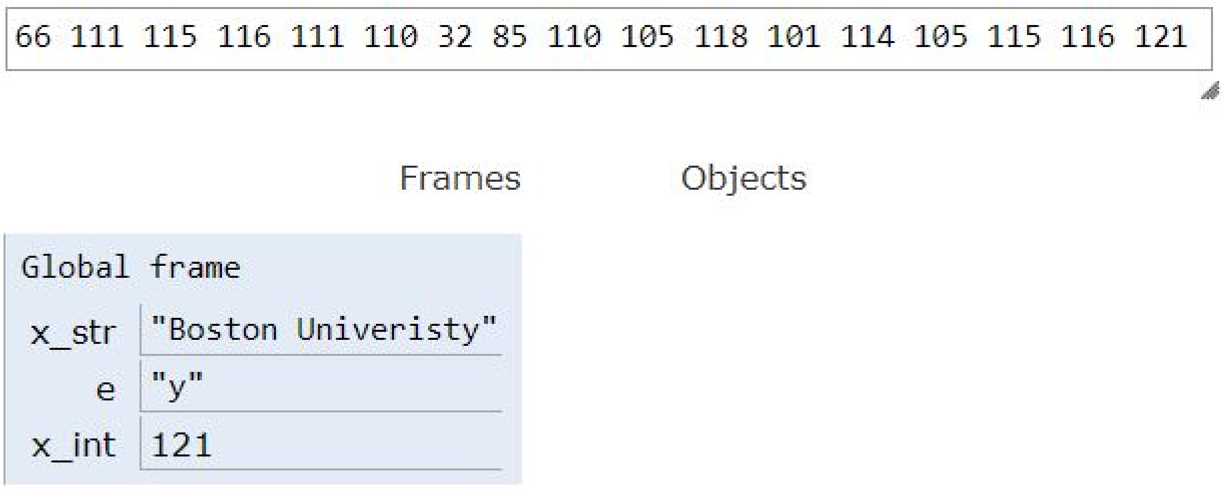
Test Yourself 3.1.03

Write a program: use *ord()* to print integer values for each character in string *x_str*:

x_str = "Boston University"

Suggested program:

```
x_str = "Boston Univeristy"
for e in x_str:
    x_int = ord(e)
    print(x_int , end = " ")
```



```
66 111 115 116 111 110 32 85 110 105 118 101 114 105 115 116 121
```

Frames Objects

Global frame

x_str "Boston Univeristy"

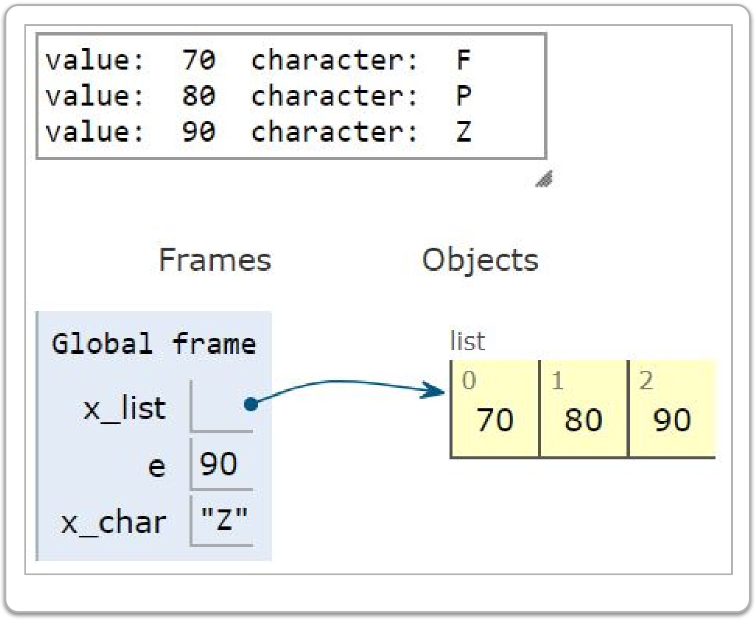
e "y"

x_int 121

chr() Function

chr(): maps integer value to corresponding character.

```
x_list = [70, 80, 90]
for e in x_list:
    x_char = chr(e)
    print('value: ', e, 'character: ', x_char)
```



```
value:  70  character:  F
value:  80  character:  P
value:  90  character:  Z
```

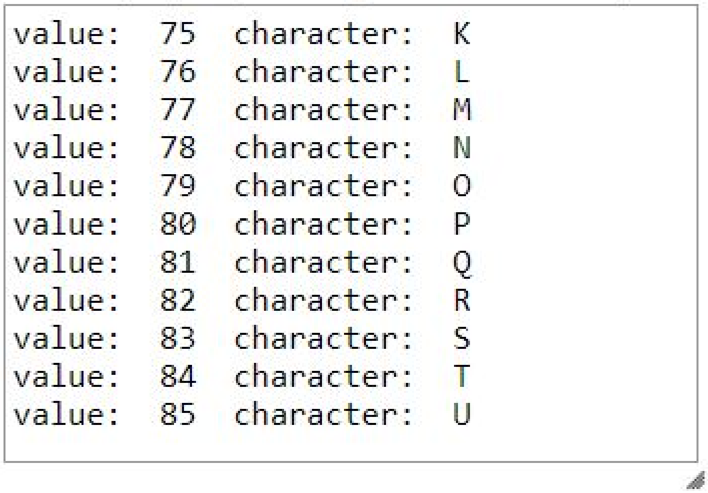
Test Yourself 3.1.04

Write a program: use *chr()* to print characters for integers from 75 to 85.

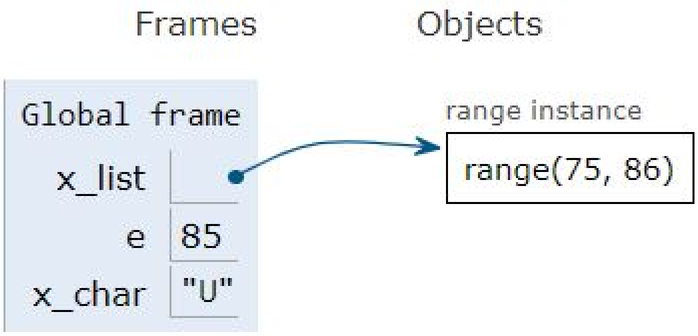
x_str = "Boston University"

Suggested program:

```
x_list = range(75, 86) # note 86, not 85
for e in x_list:
    x_char = chr(e)
    print('value: ', e, 'character: ', x_char)
```



```
value:  75  character:  K
value:  76  character:  L
value:  77  character:  M
value:  78  character:  N
value:  79  character:  O
value:  80  character:  P
value:  81  character:  Q
value:  82  character:  R
value:  83  character:  S
value:  84  character:  T
value:  85  character:  U
```



Global frame

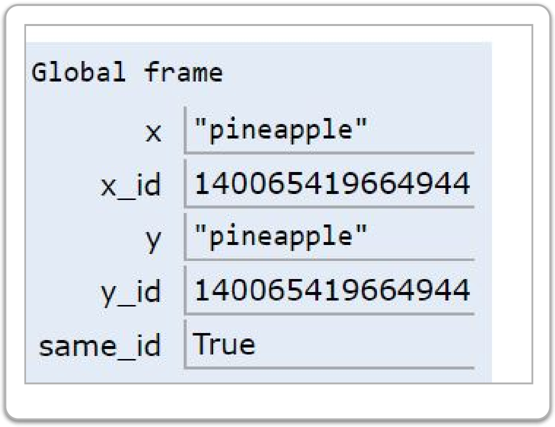
x_list	
e	85
x_char	"U"

range(75, 86)

String Immutability

Python strings are immutable.

```
x = "pineapple"
x_id = id(x)
y = 'pine' + 'apple'
y_id = id(y)
same_id = (x_id == y_id)
```

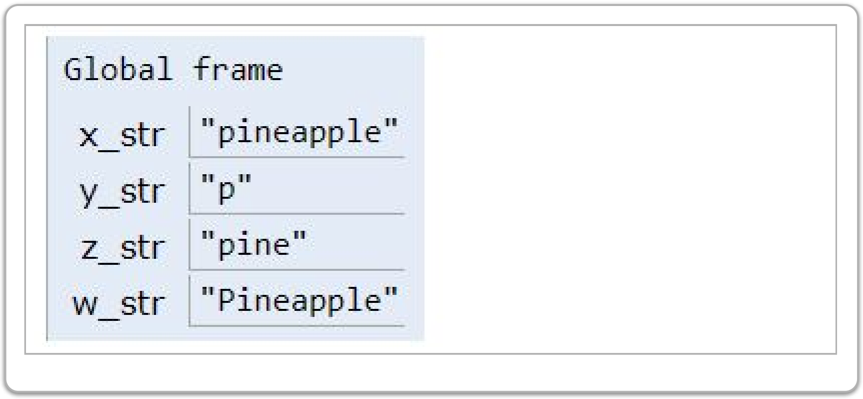


Global frame

x	"pineapple"
x_id	140065419664944
y	"pineapple"
y_id	140065419664944
same_id	True

Examples of String Methods

```
x_str = 'pineapple'
y_str = x_str[6]       # indexing
z_str = x_str[0 : 4]   # slicing
w_str = x_str.title()  # capitalize first
```



Global frame

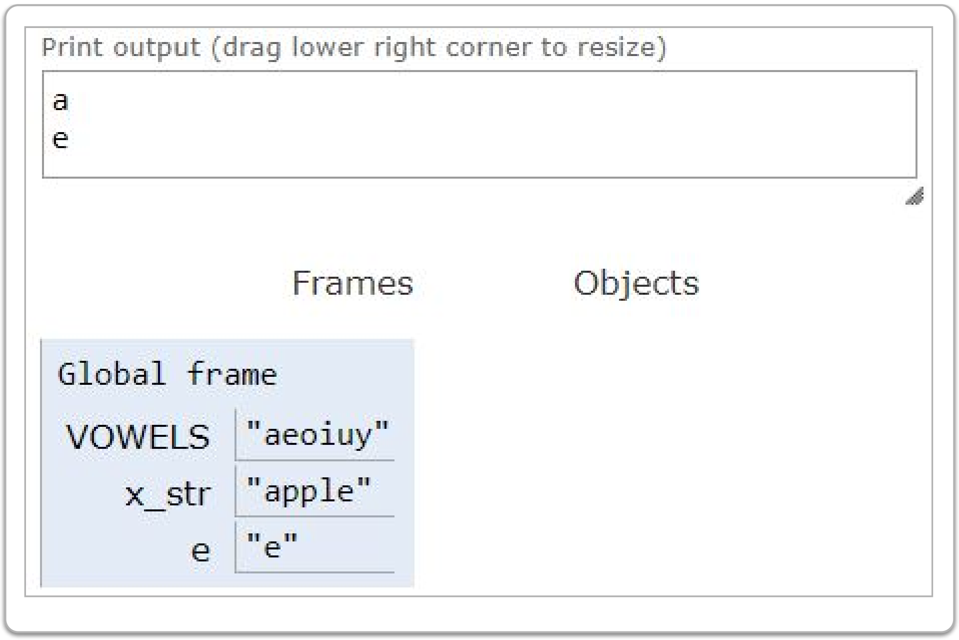
x_str	"pineapple"
y_str	"p"
z_str	"pine"
w_str	"Pineapple"

0	1	2	3	4	5	6	7	8
p	i	n	e	a	p	p	l	e

Membership & Iteration

```
# print vowels in a string
VOWELS = 'aeoiuy'
x_str = 'apple'

for e in x_str:
    if e in VOWELS:
        print(e)
```



```
a
e
```

Global frame

VOWELS	"aeoiuy"
x_str	"apple"
e	"e"

Test Yourself 3.1.05

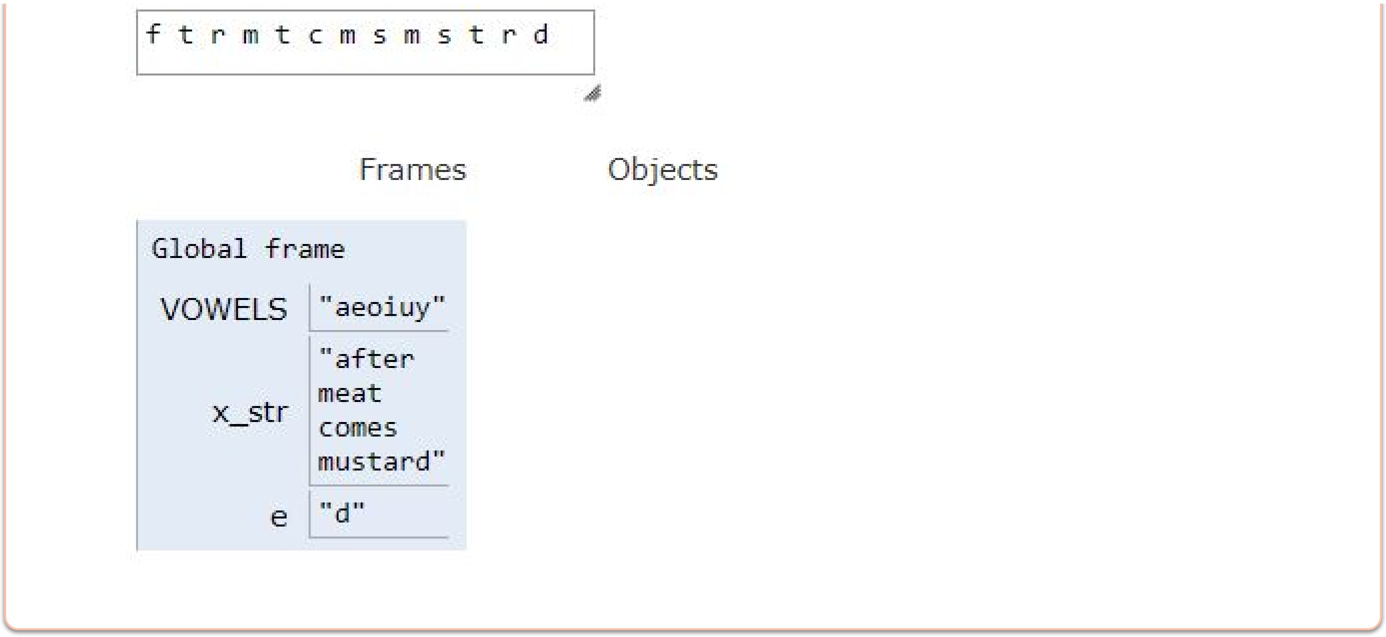
Write a program: print all consonants in string *x_str*:

"after
meat
comes
mustard"

Suggested program:

```
# print consonants in a string
VOWELS = 'aeoiuy'
x_str = """ after
meat
comes
mustard """

for e in x_str:
    if e not in VOWELS and e!= '\n':
    print(e, end = " ")
```



```
f t r m t c m s m s t r d
```

Global frame

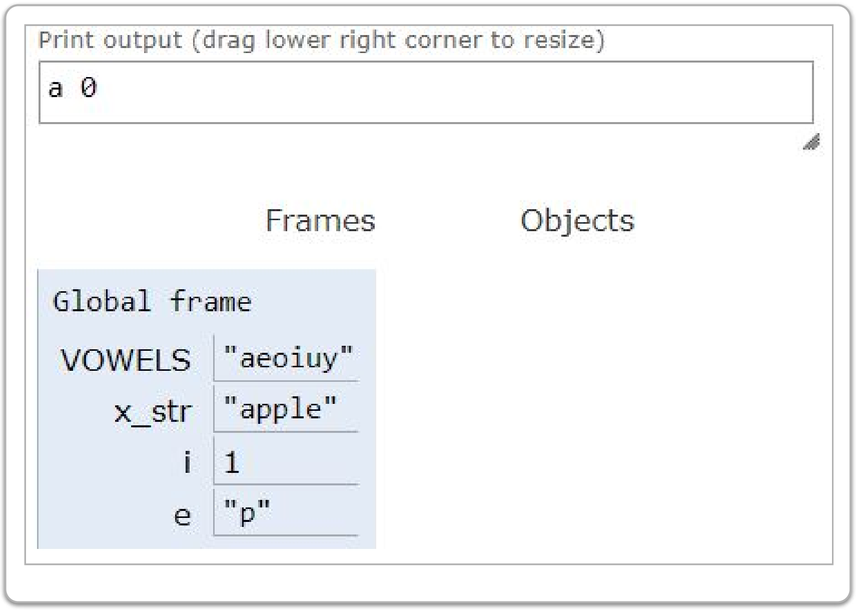
VOWELS	"aeoiuy"
x_str	"after meat comes mustard"
e	"d"

Iteration: *enumerate()*

- Using *enumerate()*, can get both index and element.
- Can use *enumerate()* in strings, lists, tuples.

```
# print vowels and positions from string
VOWELS = 'aeoiuy'
x_str = 'apple'

for i,e in enumerate(x_str):
    e = x_str[i]
    if e in VOWELS:
        print(e,i)
```



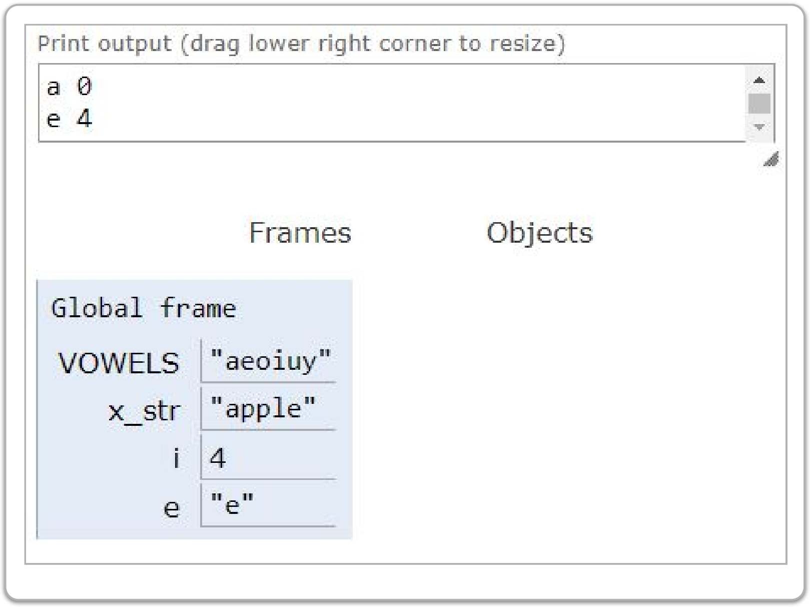
Print output (drag lower right corner to resize)

```
a 0
```

Frames Objects

Global frame

VOWELS	"aeoiuy"
x_str	"apple"
i	1
e	"p"



Print output (drag lower right corner to resize)

```
a 0
e 4
```

Frames Objects

Global frame

VOWELS	"aeoiuy"
x_str	"apple"
i	4
e	"e"

Test Yourself 3.1.06

Write a program: print all consonants and positions in string *x_str*:

"after

meat

comes

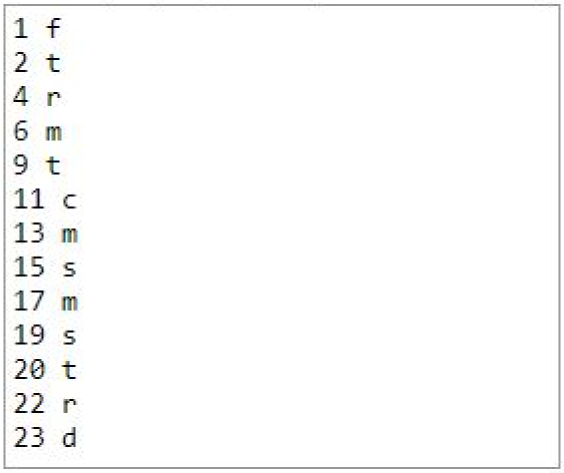
mustard"

Suggested program:

```
VOWELS = 'aeoiuy'
x_str = """ after
meat
```

```
comes
mustard """

for i, e in enumerate (x_str):
    if e not in VOWELS and e!='\n':
    print(i,e)
```



```
1 f
2 t
4 r
6 m
9 t
11 c
13 m
15 s
17 m
19 s
20 t
22 r
23 d
```

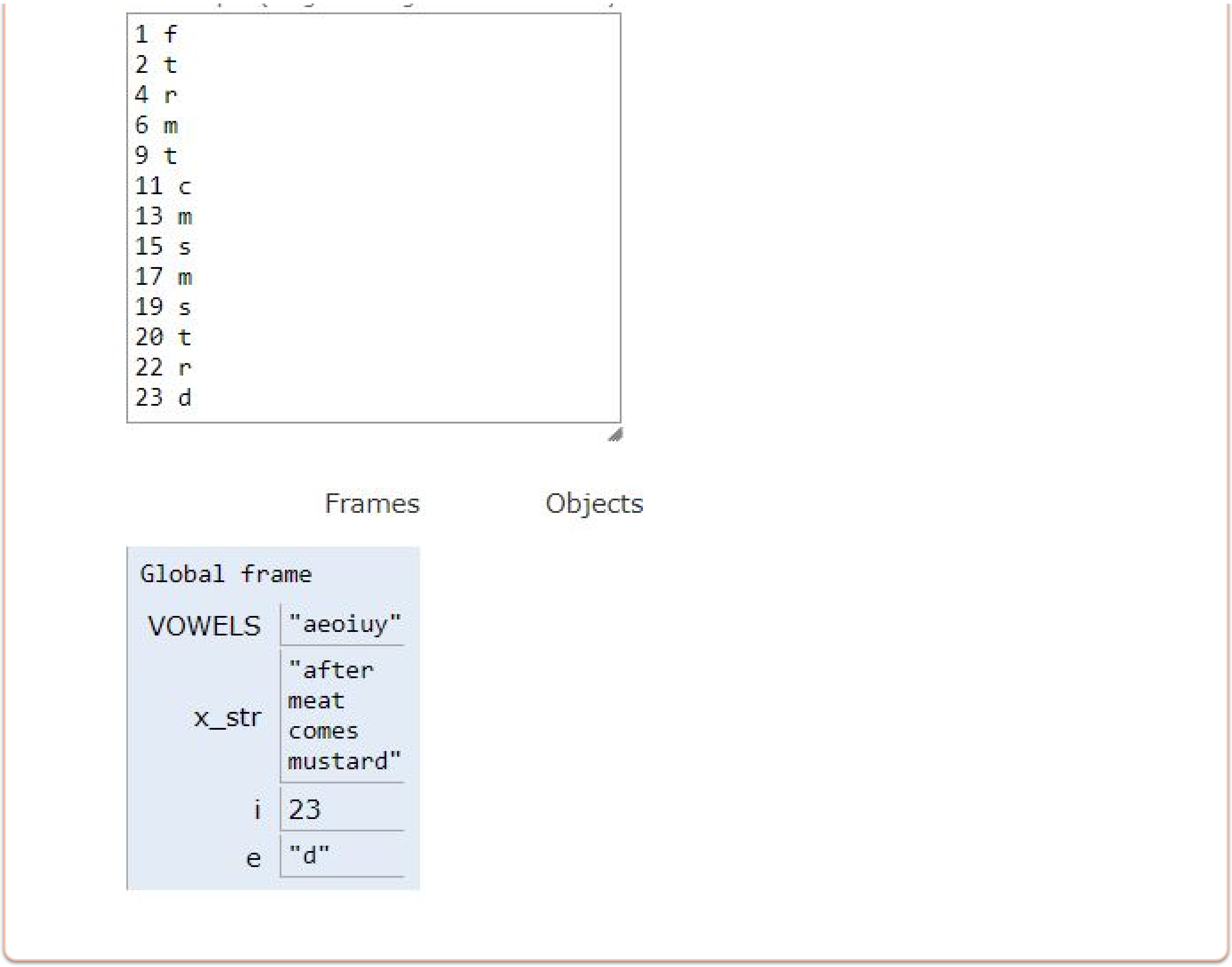
Test Yourself 3.1.07

Write a program: print vowels and positions in string x str without using *enumerate()*:

Suggested program:

```
VOWELS = 'aeoiuy'
x_str = """ after
meat
comes
mustard """

for i in range(len(x_str)):
    e = x_str[i]
    if e not in VOWELS and e!='\n':
        print(i,e)
```



```
1 f
2 t
4 r
6 m
9 t
11 c
13 m
15 s
17 m
19 s
20 t
22 r
23 d
```

VOWELS	"aeoiuy"
x_str	"after meat comes mustard"
i	23
e	"d"

Strings Functions

String Indexing

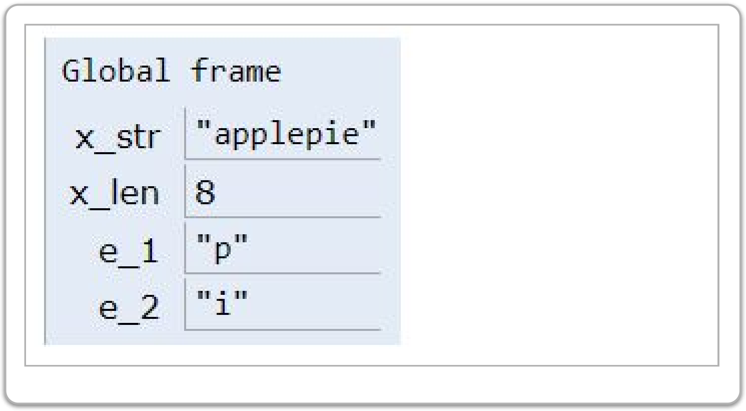
Indexing allows you to access individual characters in a string directly by using a numeric value. There are positive and negative indices.

- Positive index: String indexing is zero-based: the first character in the string has index 0, the next is 1, and so on.
- Negative index: As an alternative, Python uses **negative** numbers to index into a string: -1 means the last character, -2 means the next to last, and so on.
- Positive = negative + length

```
x_str = 'applepie'

# len(): returns the length of a string
```

```
x_len = len(x_str)
e_1   = x_str[1]
e_2   = x_str[-2]
```



Global frame

x_str	"applepie"
x_len	8
e_1	"p"
e_2	"i"

0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1

Test Yourself 3.1.08

Write a program: use positive and negative indices to extract "7" from *x_str*:

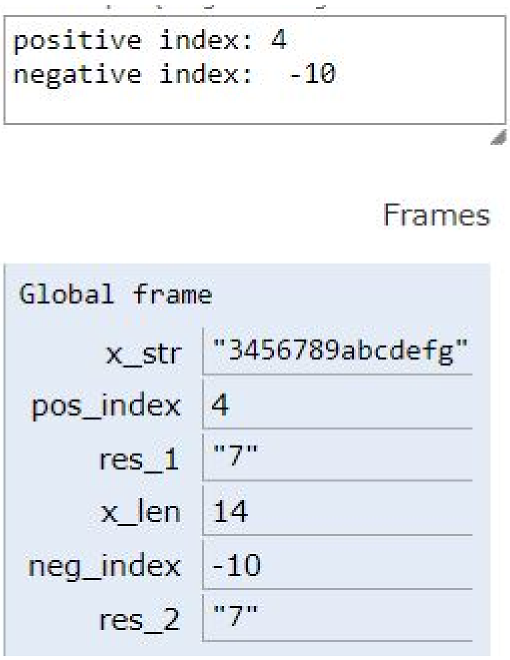
x_str = "3456789abcdefg"

Suggested program: number 7 is at index 4.

```
x_str = "3456789abcdefg"
pos_index = 4
res_1 = x_str[pos_index]

x_len = len(x_str)
neg_index = pos_index - x_len
res_2 = x_str[neg_index]

print('positive index:', pos_index)
print('negative index: ', neg_index)
```



```
positive index: 4
negative index:  -10
```

Frames

Global frame

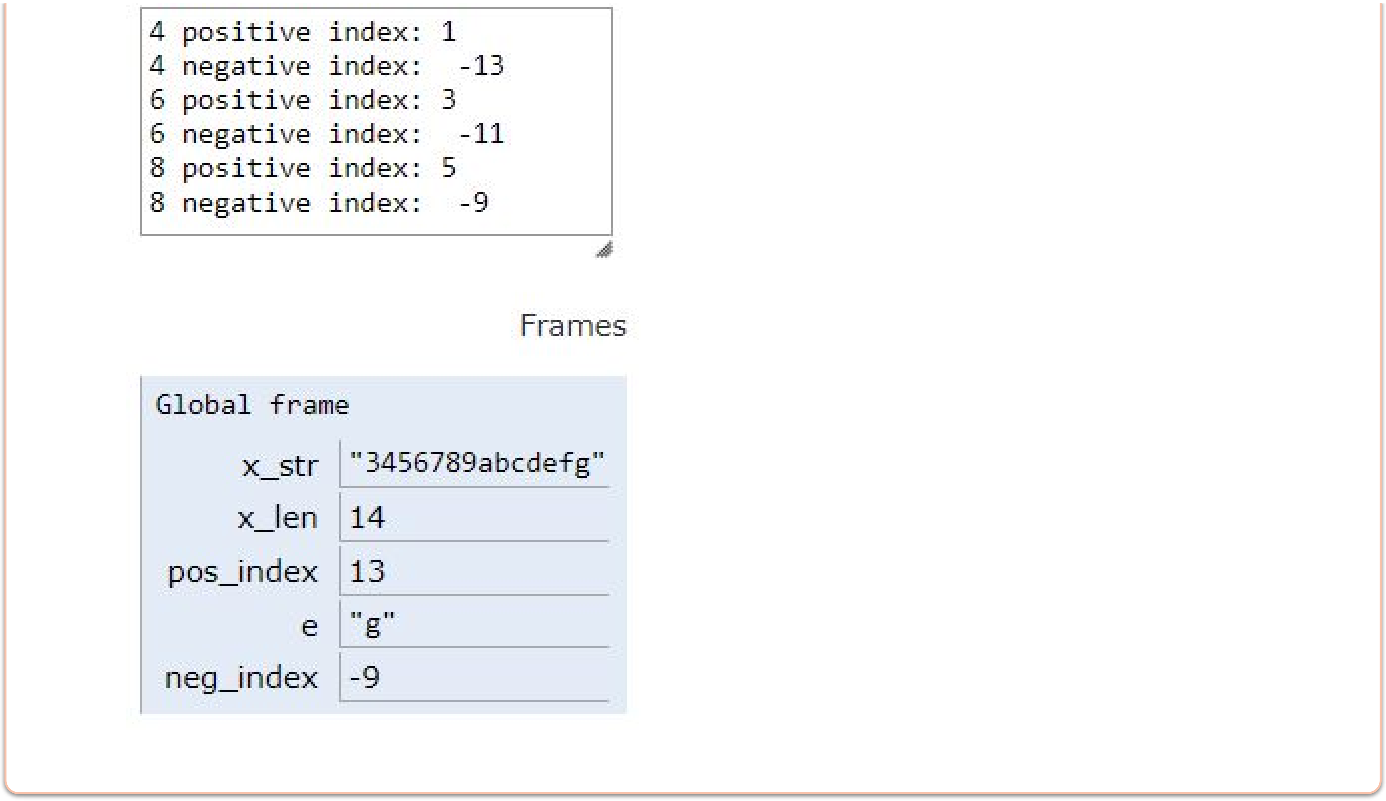
x_str	"3456789abcdefg"
pos_index	4
res_1	"7"
x_len	14
neg_index	-10
res_2	"7"

Test Yourself 3.1.09

Write a program: print positive and negative indices for even digits in *x_str*:

Suggested program:

```
x_str = "3456789abcdefg"
x_len = len(x_str)
for pos_index in range(x_len):
    e = x_str[pos_index]
    if e in "02468":
    neg_index = pos_index - x_len
    print(e, 'positive index:', pos_index)
    print(e, 'negative index: ', neg_index)
```



```
4 positive index: 1
4 negative index:  -13
6 positive index: 3
6 negative index:  -11
8 positive index: 5
8 negative index:  -9
```

Frames

Global frame

x_str	"3456789abcdefg"
x_len	14
pos_index	13
e	"g"
neg_index	-9

String Slicing

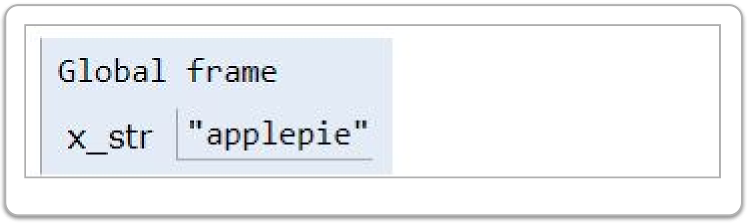
Slicing allows you to extract substrings from a string, by identifying a range of index numbers [start : end + 1 : step]:

- The first index number is where the slice starts (inclusive).
- The second index number is where the slice ends (exclusive).
- Step number: the number of characters to skip between two index numbers of a slice.
  - If the step number is 1, will take every character between two index numbers of a slice.
  - If the step number is 2, skip every other character between two index numbers.
  - If the step number is omitted, Python will default with 1.

We can use both positive and negative indices. Negative step is for reversals.

Use the string "applepie" as an example.

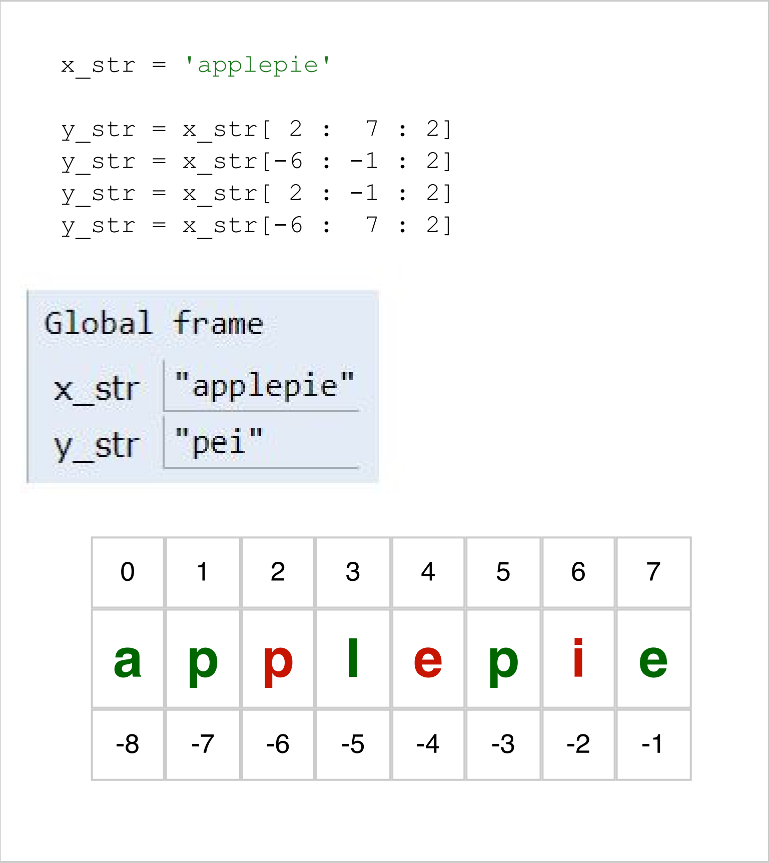
```
x_str = 'applepie'
```



Global frame

x_str	"applepie"

0	1	2	3	4	5	6	7
a	**p**	**p**	**l**	**e**	**p**	**i**	**e**
-8	-7	-6	-5	-4	-3	-2	-1



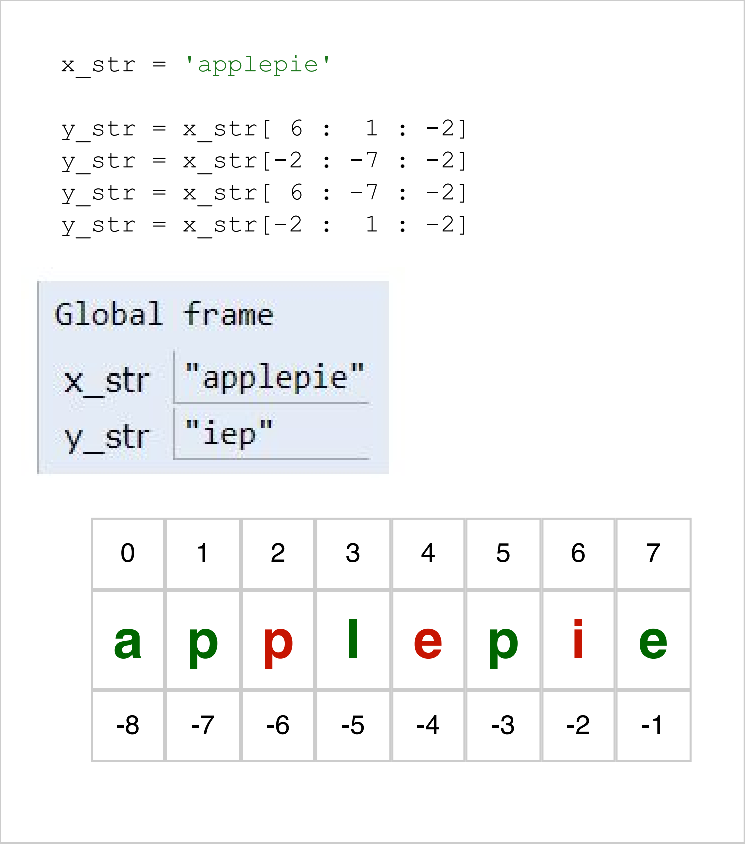
```
x_str = 'applepie'

y_str = x_str[ 2 :  7 : 2]
y_str = x_str[-6 : -1 : 2]
y_str = x_str[ 2 : -1 : 2]
y_str = x_str[-6 :  7 : 2]
```

Global frame

x_str	"applepie"
y_str	"pei"

0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1



```
x_str = 'applepie'

y_str = x_str[ 6 :  1 : -2]
y_str = x_str[-2 : -7 : -2]
y_str = x_str[ 6 : -7 : -2]
y_str = x_str[-2 :  1 : -2]
```

Global frame

x_str	"applepie"
y_str	"iep"

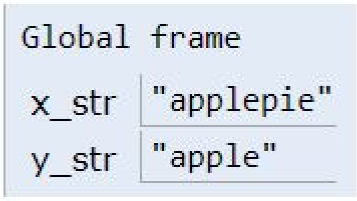
0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1

Slicing with Default

- when the first index number is omitted, Python will default with 0, the beginning of the string.
- When the the index number after the colon is omitted, Python will default to the last index, the end of the string.
- If the step number is omitted, Python will default with 1.

```
x_str = 'applepie'

y_str = x_str[0 : 5 : 1]
y_str = x_str[  : 5 : 1]  # assume defaults
y_str = x_str[  : 5]
```



Global frame

x_str "applepie"

y_str "apple"

0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1

```
x_str = 'applepie'

y_str = x_str[  : 5]
w_str = x_str[5 :  ]
```

0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1

Test Yourself 3.1.10

Write a program: show four different ways to extract "wash" from *x_str*:

x_str = "dishwasher"

Suggested program:

```
x_str = "dishwasher"
x_len = len(x_str)
a = x_str[4  : 10]
b = x_str[4  :   ]
c = x_str[-6 : 10]
d = x_str[-6 : ]
```

"Out-of-Bound" Slicing

Python string slicing handles out of range indexes gracefully—get the "largest" substring and do not produce error.

```
x_str = 'applepie'
y_str = x_str[-100 : 5]
z_str = x_str[5 : 500 ]
w_str = x_str[400 : 500]
```

Slicing vs. Indexing

```
x_str     = 'applepie'
y_slice   = x_str[4:5]
y_element = x_str[4]
z_slice   = x_str[100:101]
z_element = x_str[100]     # error
```

0	1	2	3	4	5	6	7
a	p	p	l	e	p	i	e
-8	-7	-6	-5	-4	-3	-2	-1

Test Yourself 3.1.11

Write a program: what is the result of the following slices from *x_str*:

"two plus two is four"

a. *x_str*[10]

b. *x_str*[10 : 11]

c. *x_str*[10 : 2000]

d. *x_str*[2000 : 2001]

Suggested program:

```
x_str = "two plus two is four"

a = x_str[10]
b = x_str[10 : 11]
c = x_str[10 : 2000]
d = x_str[2000 : 2001]
```

String Reversal

```
x_str = 'applepie'
y_str = x_str[ : : -1]
```

Apply string reversal to check if a string is a palindrome.

```
x_str = 'never odd or even'
y_str = x_str.replace (' ', '')
if y_str == y_str[ : : -1]:
    print(x_str, ' is a palindrome')
else :
    print(x_str, ' is not a palindrome')
```

Test Yourself 3.1.12

Write a program: reverse the string *x_str*:

"after meat comes mustard"

Suggested program:

```
x_str = "after meat comes mustard"
x_rev = x_str [ : : -1 ]
```

String *split()* Function

```
# split string using a separator
x_str = """ dogs and
cats and
bees """

x_list = x_str.split()
y_list = x_str.split(sep = '\n')
z_list = x_str.split(' and \n')
```

Test Yourself 3.1.13

Write a program: convert a string of words *x_str* into a list of words *x_list*:

> "after meat comes mustard"

Suggested program:

```
x_str  = "after meat comes mustard"
x_list = x_str.split()
```

String *join()* Function

```
# join strings in list with separator
x_list = ['dogs', 'cats', 'bees']
x_str = ','.join(x_list)
y_str = ' and'.join(x_list)
z_str = '\n'.join(x_list)
```

Test Yourself 3.1.14

Write a program: using *split()* and *join()* replace spaces with '$' in the string *x_list*:

"after meat comes mustard"

Suggested program:

```
x_str  = "after meat comes mustard"
x_list = x_str.split()
y_str  = "$".join(x_list)
```

Strings Methods

- There are many string methods (around 50) available.
- This makes Python very useful to use for text processing.

```
x = 'pineapple'
y = x.startswith("pi")
z = x.endswith("LE")
```

There are many methods to check formats.

```
x = 'pineapple'
y = x.islower()
z = x.isupper()
w = x.isdigit()
```

The isdigit() method in the following program verifies format for social security numbers.

```
x = "123-58-0089";
y = x.split("-")
valid = False
if len(y)==3:
   if (y[0].isdigit() is True) and \
      (y[1].isdigit() is True) and \
      (y[2].isdigit() is True):
          valid = True
if valid is True:
    print(x, ' is a valid ssn')
else:
    print(x, ' is not valid ssn')
```

Test Yourself 3.1.15

Write a program: verify that only numeric values are entered for a date.

x_date = "09/08/1988"

Suggested program:

```
# assume three numbers are entered
x_date = "09/08/1988"
x_list = x_date.split("/")

month_str = x_list[0]
day_str   = x_list[1]
year_str  = x_list[2]

if month_str.isdigit() is True and \
   day_str.isdigit() is True and \
   year_str.isdigit() is True:
    print(x_date, ' is a valid')
else:
    print(x_date , ' is invalid')
```

The count() method will do easy frequency counting.

```
x = 'pineapple'
y = x.count("apple")
z = x.count("e")
```

Test Yourself 3.1.16

Consider string *x_str*: "after meat comes mustard"

a. count the number of times character "*m*" appears.

b. compute position of the first character "*m*".

c. compute position of the second character "*m*".

d. replace "*mustard*" with "*dessert*".

Suggested program:

```
x_str = "after meat comes mustard"
x_count = x_str.count("m")
```

```
        first_m = x_str.index("m")

        if first_m >= 0:
            y_str = x_str[first_m + 1 : ]
            second_m = y_str.index("m")
            if second_m >=0:
                second_m = (first_m + 1) + second_m

        y_str = x_str.replace("mustard", "dessert")
```

Collections

Python Data Types

Python types are the building blocks in a language, similar to noun, verb in English.

Python has two groups of types:

1. primitive types ("atoms")
2. collections ("molecules")

There are additional special types:

1. *None* type
2. *range* type

Primitive Types

```
x_int     = 5
x_float   = 5.0
x_char    = 'a'
x_boolean = True
x_complex = 1 + 2j
```

Primitive data types also called 'atomic' data types – they are indivisible objects.

Primitive Type Method Examples

```
x_int   = 5
x_bits  = x_int.bit_length()

y_float = 0.75
y_ratio = y_float.as_integer_ratio()
```

- 'atoms' are not just values
- objects with methods

Collection Types

```
x_str    = 'pineapple'
x_list   = [1, 2, 2, 3]
x_tuple  = (1, 2, 2, 3)
x_set    = {1, 2, 2, 3} # note duplicates
x_dict   = {1: 'NY', 2: 'LA'}
```

Collection data types 'molecules' – they are complex objects.

- membership: *in/not in*
- iteration: *for*
- some ordered and/or mutable

Test Yourself 3.2.01

For each line, indicate object type (primitive or collection).

j = 5

y = 'a'

a = 2 + 2j

b = 2 + 2*j

c = [2, 2*j]

d = {j : a, y: b}

e = {j}

f = (j)

g = (j,)

Suggested answer:

```
j = 5             # primitive (integer)
y = 'a'           # primitive (character)
a = 2 + 2j        # primitive (complex number)
b = 2 + 2*j       # primitive
c = [2, 2*j]      # collection (list)
d = {j : a, y: b} # collection (dictionary)
e = {j}           # collection (set)
f = (j)           # primitive (integer)
g = (j, )         # collection (tuple)
```

Membership: *in/not in*

```
x_string = 'apple'
x_target = 'l'
if x_target in x_string:
    print(x_target , ' is in string')

y_list = ['a','p','p','l','e']
y_target = 'x'
if y_target not in y_list:
    print(y_target, ' is not in list')
```

Test Yourself 3.2.02

Construct three different collections containing words from *x_str*:

a cube has many symmetries

Suggested answer:

```
x_list  = ["a","cube","has","many","symmetries"]
x_tuple = ("a","cube","has","many","symmetries")
x_set   = {"a","cube","has","many","symmetries"}
```

Test Yourself 3.2.03

Based on the last exercise, for each collection use iteration to check if it contains the word "*many*".

Suggested program:

```
x_list  = ["a","cube","has","many","symmetries"]
x_tuple = ("a","cube","has","many","symmetries")
x_set   = {"a","cube","has","many","symmetries"}

target = "many"

for e in x_list:
    if e == target:
        print(target, ' is in list')

for e in x_tuple :
    if e == target:
        print(target, ' is in tuple')

for e in x_set:
    if e == target:
        print(target, ' is in set')
```

Test Yourself 3.2.04

For each collection use in, not in check if it contains the word "*many*".

> a cube has many symmetries

Suggested program:

```
x_list  = ["a","cube","has","many","symmetries"]
x_tuple = ("a","cube","has","many","symmetries")
x_set   = {"a","cube","has","many","symmetries"}

target = "many"

if target in x_list:
    print(target, ' is in list')
else:
    print(target, ' is not in list')

if target in x_tuple:
    print(target, ' is in tuple')
else:
    print(target, ' is not in tuple')

if target in x_set:
    print(target, ' is in set')
else:
    print(target, ' is not in set')
```

Collection Methods

```
x_string = 'pine'
x_tuple  = (1, 2, 2, 3)
x_list   = [1, 2, 2, 3]
x_tuple  = (1, 2, 2, 3)
x_set    = {1, 2, 2, 3}
x_dict   = {1: 'NY', 2: 'LA'}

y_string = x_string + 'apple'
y_tuple  = x_tuple  + (4, 5)
e_list   = x_list.pop(1)
e_set    = x_set.pop()
e_dict   = x_dict.pop(1)
```

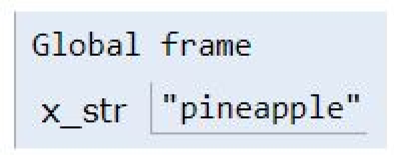
- '+' is overloaded.
- Polymorphic methods: a same function that can be applied to different types. For example, .pop() method can be applied to a list, a set, or a dictionary data type.

Python String, Tuple, and List

A Python Strings

- A Python string is an object, not just an array of character data.
- A string is ordered and immutable.
- There are many built-in methods in Python to manipulate strings.

```
x_str = 'pineapple'
```



Global frame

x_str "pineapple"

0	1	2	3	4	5	6	7	8
p	i	n	e	a	p	p	l	e

A Python List

- A Python list is an ordered collection.
- A list can contain any objects.

```
x_list = ['a','p','p','l','e']
y_list = list('apple')
z_list = ['apple', [1,2] , {1: 'NY', 2: 'LA'}]
```

A Python Tuple

- A Python tuple is an ordered collection (like list).
- A list can contain any objects.

```
x_tuple = ('a','p','p','l','e')
y_tuple = tuple('apple')
z_tuple = ('apple', [1 ,2], {1: 'NY', 2: 'LA'})
```

Strings, Lists, Tuples

- Ordered collections
- Support indexing & slicing

```
x_string = 'apple'
x_list = ['a','p','p','l','e']
x_tuple = ('a','p','p','l','e')
```

0	1	2	3	4
a	p	p	l	e

Test Yourself 3.2.05

Show two different ways to construct *x_list* and *x_tuple* from *x_str*:

x_str = "apple"

Suggested program:

```
x_str = "apple"
x_list_1  = list(x_str)
x_tuple_1 = tuple(x_str)

x_list_2 = list()
for e in x_str:
    x_list_2 = x_list_2 + [e]

x_tuple_2 = tuple()
for e in x_str :
    x_tuple_2 = x_tuple_2 + (e, )
```

Python Set and Dictionary

A Python Set

- A Python set is un-ordered, unique elements.
- There are restrictions on elements.

```
x_set = {'a','p','p','l','e'}
y_set = set('apple')
z_set = set()
```

A Python Dictionary

- A Python dictionary is a collection of (key, value) pairs.
- Such pairs are called items.
- There are built-in functions for keys, values and items.

```
x_dict = {1: 'NY', 2: 'LA'}
target_key = 1
target_value = x_dict[target_key]
```

Iteration in Collections

```
VOWELS = 'aeoiuy'
x_string = 'apple'
x_list   = ['a','p','p','l','e']
x_tuple  = ('a','p','p','l','e')
x_set    = {'a','p','p','l','e'}

for e in x_string:
    if e in VOWELS:
        print(e, end='')

for e in x_list:
    if e in VOWELS:
        print(e, end='')

for e in x_tuple:
    if e in VOWELS:
        print(e, end='')

for e in x_set:
    if e in VOWELS:
        print(e, end='')
```

Test Yourself 3.2.06

Write iterations to print consonants from the following collections:

```
x_str   = "automobile"
x_list  = list(x_str)
x_tuple = tuple(x_str)
x_set   = set(x_str)
```

Suggested program:

```
x_string = 'automobile'
VOWELS = 'aeoiuy'

x_list  = list(x_str)
x_tuple = tuple(x_str)
x_set   = set(x_str)

print(" from list: ")
for e in x_list:
    if e not in VOWELS:
        print(e, end='')
print("\n from tuple: ")
for e in x_tuple:
    if not e in VOWELS:
        print(e, end='')

print("\n from set: ")
for e in x_set:
    if not e in VOWELS:
        print(e, end='')
```

Test Yourself 3.2.07

Based on the last exercise, why does *x_set* contain one less element than the other three collections?

Suggested answer: we have two characters "o", only one will be in the set.

Ordered Collections

- Enumerate: strings, lists and tuples are ordered collections.
- Support indexing & slicing.

```
x_string = 'apple'
x_list = ['a','p','p','l','e']
x_tuple = ('a','p','p','l','e')
```

0	1	2	3	4
a	p	p	l	e

enumerate() in Collections

- Can use *enumerate()* in ordered collections only: strings, lists, tuples.

```
# print vowels and positions from collections
VOWELS = 'aeoiuy'
x_string = 'apple'
x_list   = ['a','p','p','l','e']
x_tuple  = ('a','p','p','l','e')

for i,e in enumerate(x_string):
    e = x_string[i]
    if e in VOWELS:
        print(e, i)

for i,e in enumerate(x_list):
    e = x_list[i]
    if e in VOWELS:
        print(e, i)

for i,e in enumerate(x_tuple):
    e = x_tuple[i]
    if e in VOWELS:
        print(e, i)
```

Test Yourself 3.2.08

Use *enumerate()* iteration to print consonants and their positions from the following collections:

```
x_str   = "automobile"
x_list  = list(x_str)
x_tuple = tuple(x_str)
```

Suggested program:

```
# print non - vowels and positions
VOWELS = 'aeoiuy'

x_str   = 'automobile'
x_list  = list(x_str)
x_tuple = tuple(x_str)

print(" from string: ")
for i,e in enumerate(x_str):
    if e not in VOWELS:
        print(e,i, end=" ")

print("\n from list: ")
for i,e in enumerate(x_list):
    if e not in VOWELS:
        print(e,i, end=" ")

print("\n from tuple: ")
for i,e in enumerate(x_tuple):
    if e not in VOWELS:
        print(e,i, end=" ")
```

Indexing in Collections

```
x_string = 'apple'
x_list   = ['a','p','p','l','e']
x_tuple  = ('a','p','p','l','e')
e_1      = x_string[1]
e_2      = x_list[1]
e_3      = x_tuple[1]
```

0	1	2	3	4
a	p	p	l	e

Slicing in Collections

```
x_string = 'apple'
x_list   = ['a','p','p','l','e']
x_tuple  = ('a','p','p','l','e')
y_string = x_string[1 : 4]
y_list   = x_list[1 : 4]
y_tuple  = x_tuple[1 : 4]
```

Mutability

Mutable collections:

- lists
- set
- dictionary

Immutable collections:

- strings
- tuples

List Mutability

```
x_list = ['a','p','p','l','e']
x_id   = id(x_list)
```

```
x_value = x_list.pop(-1)  # remove last
x_id    = id(x_list)
```

Set Mutability

```
x_set = {'a','p','p','l','e'}
x_id   = id(x_set)
```

```
x_value = x_set.pop()  # remove random
x_id    = id(x_set)
```

Dictionary Mutability

```
x_dict = {1: 'NY', 2: 'LA', 3: 'SF'}
x_id   = id(x_dict)
```

```
x_value = x_dict.pop(2)  # remove key = 2
x_id    = id(x_dict)
```

Summary of Collections

Collection	Ordered	Mutable
string	yes	no
list	yes	yes
tuple	yes	no
set	no	yes
dictionary	no	yes

There are some variations:

1. "frozen" set (immutable)
2. ordered dictionary

Test Yourself 3.2.09

Is it possible to convert an immutable collection to a mutable one with same elements?

Suggested program:

```
# it is possible: tuple → list
x_tuple = (1, 2, 3)
x_list  = list(x_tuple)
```

Test Yourself 3.2.10

Is it possible to convert a mutable collection to an immutable one with same elements?

Suggested program:

```
# it is possible: list → tuple
x_list = [1, 2, 3]
x_tuple  = tuple(x_list)
```

Common Methods - *clear()*, *copy()*, *count()*, and *index()*

Method	str	list	tuple	set	dict
clear	no	yes	no	yes	yes
copy	no	yes	no	yes	yes
count	yes	yes	yes	no	no
index	yes	yes	yes	no	no
pop	no	yes	no	yes	yes
remove	no	yes	no	yes	no
update	no	no	no	yes	yes

1. Many polymorphic methods
2. Ordered collections: string, list, tuple
3. Mutable collections: list, set, dictionary

Collections: *clear()*

- The *clear()* method applies to mutable collections.
- The *clear()* method is done in place.

```
x_list = ['a','p','p','l','e']
y_set  = {'a','p','p','l','e'}
z_dict = {1: 'NY', 2: 'LA'}
```

```
x_list.clear()
y_set.clear()
z_dict.clear()
```

Collections: *copy()*

```
x_list = ['a','p','p','l','e']
y_set  = {'a','p','p','l','e'}
z_dict = {1: 'NY', 2: 'LA'}
```

```
x_copy = x_list.copy()
y_copy = y_set.copy()
z_copy = z_dict.copy()
```

Collections: *count()*

The *count()* method counts number of occurencies.

```
x_str   = 'apple'
y_list  = ['a','p','p','l','e']
z_tuple = ('a','p','p','l','e')
```

```
x_count = x_str.count('p')
y_count = y_list.count('p')
z_count = z_tuple.count('p')
```

Test Yourself 3.2.11

Count the number of occurrences of "y" in

x_str="monday tuesday"

Suggested program:

```
x_str = "monday tuesday"
target = "y"
x_count = x_str.count(target)
print(target, ' occurs ', x_count, 'times')
```

Test Yourself 3.2.12

For each letter construct a dictionary of frequency counts:

x_str="monday tuesday"

Suggested program:

```
x_str = "monday tuesday"
x_dict = dict()

for e in x_str:
    if e not in x_dict.keys():
        x_dict[e] = 1
    else:
        x_dict[e] = x_dict[e] + 1
```

Collections: *index()*

- The *index()* method applies to ordered collections.
- The *index()* method indexes of first ocuurency.
- Note: value must exist.

```
x_str   = 'apple'
y_list  = ['a','p','p','l','e']
z_tuple = ('a','p','p','l','e')
```

```
x_count = x_str.index('e')
y_count = y_list.index('p')
z_count = z_tuple.index('l')
```

Test Yourself 3.2.13

compute the position of first "a" in

x_str="monday tuesday"

Suggested program:

```
x_str = "monday tuesday"
x_count = x_str.count("a")
```

Test Yourself 3.2.14

Compute the position of second "a" in the same string

x_str="monday tuesday"

Suggested program:

```
x_str = "monday tuesday"
first = x_str.index("a")
if first >= 0:
    y_str = x_str[first + 1 :]
    second = y_str.index("a")
    if second >=0:
        second = (first + 1) + second
```

Boston University Metropolitan College