CS1302 - Lecture 8 Lists and Tuples (Sequence_Types.ipynb)
(Dr Helena WONG)

==========

[Helena's supplements]

1. Motivation of Composite data type

Example: average_five_numbers
How to store a sequence of items in Python
What is the difference between tuple and list
mutable vs immutable

2. Constructing sequences

How to create tuple/list? **(1) Enclosure, (2) Comprehension**

[Enclosure]
How to create a tuple/list by enumerating its items?
- **Tuple of one element: (0,)** - "," is needed
  otherwise "()" means grouping some terms in an expression
- *** for unpacking an iterable object**

[Comprehension]
How to use a rule to construct a tuple/list
- Comprehension has three parts:
  output_expression, iteration, conditional filtering, or
  output_expression with condition, iteration, conditional filtering
- Examples:

```
list1 = [ i for i in range(1,10) if i%2==0]
prices = [ i if i > 3.2 else 0 for i in [1.25, 9.45, 10.22, 3.78, 5.92, 1.16] ]
```

- Biased coin flips, Estimate the bias, **sum** and **len**

3. Selecting items in a sequence

How to traverse a tuple/list
Some functions for list and tuples: **zip**, **reversed**
How to select an item in a sequence (subscription)
How to select multiple items?
Quick Sort

4. Mutation and Aliasing

5. Different methods to operate on a sequence

Builtin's **sorted** function vs **our quicksort**
Attributes of tuple vs list, **in** and **not in**
Common attributes: **count, index**
Tuple-specific attributes: **none**
List-specific attributes:
append, clear, copy, extend,
insert, pop, remove, reverse, sort
*** copy returns the new resultant list, others mutate the original list**

.reverse() and reversed()
.copy()
.sort() and sorted()
.extend(), .append(), .insert()
.pop(), .remove(), .clear(),
del ___ # a selection of a list

==========

==========

Start-up / Discussion activities:

==========

[Start-up / Discussion Exercise A]

Which one is valid? (I) or (II)?

```
def f1(a):
    print(a)

f1((3, 4, 5)) # (I)
f1(3, 4, 5)   # (II)
```

[Start-up / Discussion Exercise B]

The unpacking operator * : unpack an iterable object

Explain what happens:

```
print(range(2))   # output _______
print(*range(2))  # output _______
print('apple')    # output _______
print(*'apple')   # output _______
print('a', 'p', 'p', 'l', 'e') # output _______
```

[Start-up / Discussion Exercise C] Basics of tuples and lists

(I) **() is for tuple. [] is for list.**

```
data1 = (3,3,3)
data2 = [3,3,3]
print(data1, type(data1), data2, type(data2), sep="\n")
```

More (i): Many useful functions that work with iterables apply to tuples and lists

```
# help(min)
# help(max)

# help(all)
# all(iterable): return True if all items in the iterable are considered True

# help(any)
# any(iterable): return True if any item in the iterable is considered True

print(min([3,10,34,5])) # 3
print(max([3,10,34,5])) # 34

print(all([0,10,34,5])) # False
print(all([1,10,34,5])) # _________
print(all([True,False,True])) # _________
print(all([True,True])) # _________
print(all([])) # _________
print(all(["apple", "", "pear"])) # _________

print(any(["apple", "", "pear"])) # _________
```

More (ii): Selection

Subscription: pick one

```
data1 = (3,30,300)
data2 = [3,30,300]
print(data1[1], data2[-1])
```

Slicing:

```
data1 = (3,30,300,3000,30000)
print(data1[3:5])
print(data1[:])
```

More (iii): Lists are mutable. Tuples and Strings are not.

```
string1 = "hello"
data1 = ('a','b','c')
data2 = ['a','b','c']

print(data1[0], data2[0], string1[0]) # a a h
data2[0] = 'x' # also work for data1[0], string[1]?  NO!
```

More (iv): Creating a List/Tuple from other iterables

```
# Constructor functions (like str() etc.): list(), tuple()
# unpacking operator *
s = "hello"
lst1 = list(s)
lst2 = [*s]

r = range(1000)
lst3 = list(r)
lst4 = [*r]
print(len(lst3))
print(lst3[:10],"..",lst3[-10:])
```

```
s = "hello"
t1 = tuple(s)
t2 = (*s,)

r = range(1000)
t3 = tuple(r)
t4 = (*r,)
print(len(t3))
print(t3[:10],"..",t3[-10:])
```

(II) () is for tuple. Note the additional commas (,)

When () is not for tuple:

x1 = 2*3+4

x2 = (2*3)+4

x3 = 1302

x4 = (1302)

```
x1 = 2*3+4
x2 = (2*3)+4

x3 = 1302
x4 = (1302)
```

x1	10
x2	10
x3	1302
x4	1302

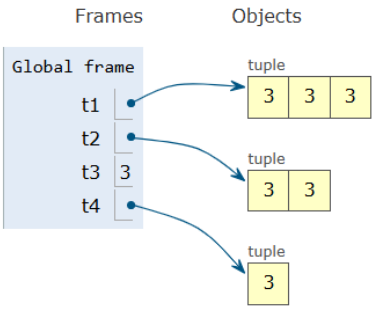
```
#--------------------

t1 = (3,3,3)  # a tuple?
t2 = (3,3)    # a tuple?
t3 = (3)      # a tuple?
t4 = (3,)     # a tuple?
```

```
1  t1=(3,3,3)  # a tuple?
2  t2=(3,3)    # a tuple?
3  t3=(3)      # a tuple?
4  t4=(3,)     # a tuple?
5
```

line that just executed

next line to execute

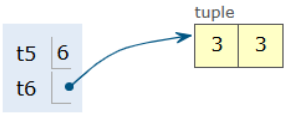



```
#--------------------

t5 = (1+2)*2 # gives 6?

t6 = (1+2,)*2 # gives 6?
```

```
1  t5=((1+2)*2) # gives 6?
2  t6=((1+2,)*2) # gives 6?
3
4
```



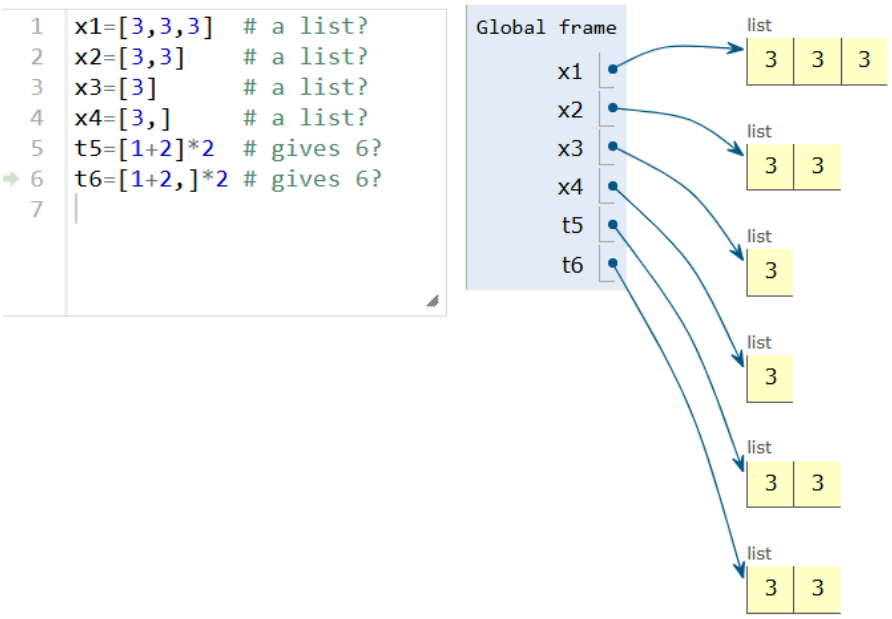
Explanation:

The parentheses in (1+2) is used to indicate the addition needs to be performed first, but the the parentheses in (1+2,) creates a tuple.

(III) **[] is for list. Need additional commas (,)?**

Question: How about []?

```
x1=[3,3,3]  # a list?
x2=[3,3]    # a list?
x3=[3]      # a list?
x4=[3,]     # a list?
t5=[1+2]*2  # gives 6?
t6=[1+2,]*2 # gives 6?
```



(IV) **Slicing**

```
a = (*range(10),)
print(a)

a = ['a','b','c','d','e','f','g','h','i','j']
print(a)
print(a[:4])     # start defaults to 0
print(a[1:])     # stop defaults to len(a)
print(a[1:4:]) # step defaults to 1: a[1:4:1]

a = ['a','b','c','d','e','f','g','h','i','j']
print(a[-1:]) # _____
print(a[:-1]) # _____
print(a[::-1]) # _____
```

Digest it! (1) id, mutation, +=, +

```
s1 = (*"Hello",) # (i) tuple
print(id(s1), s1)
s1 += ("!",)
print(id(s1), s1)

s2 = [*"Hello"] # (ii) list
print(id(s2), s2)
s2 += ["!"]
print(id(s2), s2)
```

output:

```
18495120 ('H', 'e', 'l', 'l', 'o')
21178360 ('H', 'e', 'l', 'l', 'o', '!')
18967456 ['H', 'e', 'l', 'l', 'o']
18967456 ['H', 'e', 'l', 'l', 'o', '!']
```

Observe the result! Explanation:
(i) creates a new object
(ii) does not create a new object

```
# ------------------
# Pitfall: xx += yy is not the same as xx = xx + yy
# xx = xx + yy will anyway create a new object

s1 = (*"Hello",)
print(id(s1), s1)
s1 = s1 + ("!",)
print(id(s1), s1)

s2 = [*"Hello"]
print(id(s2), s2)
s2 = s2 + ["!"]
print(id(s2), s2)
```

output:
18495120 ('H', 'e', 'l', 'l', 'o')
12358280 ('H', 'e', 'l', 'l', 'o', '!')
21273448 ['H', 'e', 'l', 'l', 'o']
20902736 ['H', 'e', 'l', 'l', 'o', '!']

Digest it! (2) id, mutation, +=, +

```
# List example
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]
prices=[]
print("id is: ", id(prices), "content is", prices)

for i in original_prices:
    if i>3:
        prices += [i]    # python "extends" (find new memory, copy to there) the memory if not enough
    else:
        prices += [0]
    print("id is: ", id(prices), "content is", prices)

# Tuple example
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]
prices = ()
print("id is: ", id(prices), "content is", prices)

for i in original_prices:
    if i>3:
        prices += (i,)    # prices is assigned for a new resultant list
    else:
        prices += (0,)
    print("id is: ", id(prices), "content is", prices)
```

[Start-up / Discussion Exercise D]

Explain what happens:

```
my_list = [1, 2, False, "abc", complex(1, 2)]
my_list[4] = complex(3, 4)
my_list.append(99)
```

#.append is the same as __ += __ ?
#.append is the same as __ = __ + __ ?

```
my_list = [1, 2]
copy1 = my_list
copy2 = my_list
copy3 = my_list
copy1.append(88)
copy2 += [123]
copy3 = copy3 + [456]
```

The following shows that tuple is immutable (items cannot change, not extensible).
Explain what happens:

```
my_tuple = (1, 2, False, "abc", complex(1, 2))
# my_tuple[4] = complex(3, 4) # not allowed
# my_tuple.append(99) # not allowed
```

[Start-up / Discussion Exercise E]

Explain what happens:

```
a = (*range(5),)
print(a)
b = reversed(a)
print(*b) # just like print(4,3,2,1,0)

a = (*range(5),)
print(a)
b = reversed(a)
print((*b,))
```

```
# Challenge (1): What are the outputs?
a = (*range(5),)
b = reversed(a)
print(type(b)) # output: _____
print((*b,)) # output: _____
print((*b,)) # output: _____
help(b)
```

```
# Challenge (2):    Which one is mutation?    Which one is aliasing?
a = ["Ada", "Brian"]
r = reversed(a) #creates a reverse obj for a's list
a = ["Peter", "Queenie"] # mutation? aliasing?
print(*r) # Brian Ada

a = ["Ada", "Brian"]
r = reversed(a) #creates a reverse obj for a's list
a[:] = ["Peter", "Queenie"] # mutation? aliasing?
print(*r) # Queenie Peter
```

```
# aliasing: assign a variable to be an alias for an object
# mutation: change the contents in an object
```

==========
Comprehension and Examples
==========
- There are two forms:

(I) output_expression, iteration, [conditional filtering], or

list1 = [i for i in range(1,10) if i%2==0]

prices1 = [i for i in [1.25, 9.45, 10.22, 3.78, 5.92, 1.16] if i<10]

(II) output_expression with condition, iteration, [conditional filtering]

prices2 = [i if i > 3.2 else 0 for i in [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]]

prices3 = [i if i > 3.2 else 0 for i in [1.25, 9.45, 10.22, 3.78, 5.92, 1.16] if i<10]

```
Testing:
print(prices1)
print(prices2)
print(prices3)

Output:
[1.25, 9.45, 3.78, 5.92, 1.16]
[0, 9.45, 10.22, 3.78, 5.92, 0]
[0, 9.45, 3.78, 5.92, 0]
```

```
#if we want to decide the output for each original value, we can put if conditional in front of for loop
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]
prices = [i if i > 3.2 else 0 for i in original_prices]
print(prices)
```

```
#the above is a comprehension, it's equivalent to the following for loop
prices=[]
for i in original_prices:        #the iterator
    if i>3.2: #conditional filtering
        prices = prices + [i]
    else:
        prices = prices + [0]
print(prices)
```

```
#note the following codes are wrong
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]
prices = [i if i > 3.2 for i in original_prices]    #if there's only if statement, do you mean filtering (ignore some inputs)?
prices = [i for i in original_prices if i > 3.2 else 0] #if there's a if ...else statement, do you mean deciding output for each input?

(i) Extract big prices (less inputs obtained):
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]

big_prices = [i for i in original_prices    if i>3]
print(big_prices)

big_prices = [i if i>3 for i in original_prices]    #error
print(big_prices)

(ii) Change low prices to zero (same six prices in the output)
original_prices = [1.25, 9.45, 10.22, 3.78, 5.92, 1.16]

chg_low_to_zero = [i for i in original_prices if i > 3.2 else 0] # error
chg_low_to_zero = [i if i>3 else 0 for i in original_prices]

# igore expensive things
chg_low_to_zero_v2 = [i if i>3 else 0 for i in original_prices if i<10]
chg_low_to_zero_v2
```

```
#------------------------
#tuple?
a = tuple(x**2 for x in range(10)) # Use the tuple constructor
print(a)
b = (x**2 for x in range(10)) # can this line generate a tuple? No, it generates a generator
print(b)

#how to generate a tuple using parenthesis?
c=(*(x**2 for x in range(10)),)
print(c)
```

```
5]: a = tuple("helena")
    print(a)

    ('h', 'e', 'l', 'e', 'n', 'a')

6]: a = tuple([1,3,9])
    print(a)

    (1, 3, 9)

7]: a = tuple(x**2 for x in range(10))
    print(a)

    (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

8]: a = [x**2 for x in range(10)]
    print(a)

    [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

2]: a = (x**2 for x in range(10)) # a generator object!
    print(a)

    <generator object <genexpr> at 0x7f85b3ded850>

4]: a = (   * ( x**2 for x in range(10) ) , )
    print(a)

    (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

```
#------------------------
```

==========

The Coin flip example

==========

```
from random import random as rand
print(rand(),rand(),rand())
p = 0.5
coin_flips = ['H' if rand() <= p else 'T' for i in range(10000)]
        coin_flips=[]
        for i in range(1000):
            if rand()<=p:
                coin_flips=coin_flips+['H']
            else:
                coin_flips=coin_flips+['T']

print('Coin flips:',*coin_flips[:10] , "...", *coin_flips[9990:])

p = 0.85 # change to get head (not fifty-fifty)

head_ind = [1 if outcome == 'H' else 0 for outcome in coin_flips]
# print(head_ind)
print(*head_ind[:10] , "...", *head_ind[9990:])

sum(head_ind), sum(head_ind)/len(head_ind)
```

Exercise 4 in Sequence_Types.ipynb: **variance(seq)**

==========

Zip

==========

```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica")

for i in zip(a,b):
    print(i)

for p1,p2 in zip(a,b):
    print(p1,p2)
```

==========

The Quicksort program (revised for tracing)

==========

```
def quicksort(seq):
    if len(seq) <= 1:
        return list(seq)

    i = len(seq)//2 # let's pick the middle

    pivot = seq[i]
    others = [*seq[:i], *seq[i + 1:]]

    left = [x for x in others if x < pivot]
    right = [x for x in others if x >= pivot]
    left = quicksort(left)
    right = quicksort(right)

    return [*left, pivot, *right]

seq = [8,5,6,10,3]
result = quicksort(seq)

# seq = [6,5,4,6,10,3]
result = quicksort(seq)
```

==========

Explanation of Aliasing and Mutation

aliasing: assign a variable to be an alias for an object
mutation: change the contents in an object

==========

aliasing:
- name an (created / existing) object
- e.g.
```
a = ('a','b','c')
b = ['d','e','f']
a = b
```

mutation:
- change content of existing obj
- not creating new object
- e.g.
```
a[0]='x'
b[::]=['s','t']
```

More example (I)

```
b =  [1,1,1,1,1,1,1,1]  # aliasing
b[::2] =  [1,2,3,4] # multation
```

The first assignment **b =** [1,1,1,1,1,1,1,1] is aliasing,
which gives the list the target name/identifier b.

The second assignment **b[::2] =** [1,2,3,4] is mutation,
which changes members in the list.
note: the target b[::2] is not an identifier.

More example (II)

```
b = [*range(10)]    # aliasing
b[::2] = b[:5]
print(b)
b[0:1] = b[:5]
print(b)
print(len(b[::2]))
print(len(b[:5]))
b[::2] = b[:5]    # fail: attempt to assign sequence of size 5 to extended slice of size 7
```

==========

Tuple/List functions - sorted

==========

```
import random
seq = [random.randint(0, 99) for i in range(10)]
sorted(seq)
```

```
%%timeit
quicksort(seq)

%%timeit
sorted(seq)
```

Quora

Atto a	10^{-18}	0.000 000 000 000 000 001
Femto f	10^{-15}	0.000 000 000 000 001
Pico p	10^{-12}	0.000 000 000 001
Nano n	10^{-9}	0.000 000 001
Micro µ	10^{-6}	0.000 001
Milli m	10^{-3}	0.001
Centi c	10^{-2}	0.01
Deci d	10^{-1}	0.1
	10^{0}	1
Deca da	10^{1}	10
Hecto h	10^{2}	100
Kilo k	10^{3}	1 000
Mega M	10^{6}	1 000 000
Giga G	10^{9}	1 000 000 000
Tera T	10^{12}	1 000 000 000 000

==========

Tuple/List: The in operator and not in

==========

```
tuple1=('apple','banana')
print('apple' not in tuple1)
print('cherry' in tuple1)
```

```
a1, a2 = dir(list), dir(tuple)
c = [a for a in a1 if a in a2 and a[0]!="_"]
print(c) # ['count', 'index']
    # https://stackoverflow.com/questions/14671487/what-is-the-difference-in-python-attributes-with-underscore-in-front-and-back
    # _single_leading_underscore: weak "internal use" indicator. E.g. from M import * does not import _...
```

==========

Tuple/List methods - count and index

==========

```
a = (1,2,2,4,5)
print(a.count(2))
print(a.index(2))
```

==========

List method - reverse (different from reversed)

==========

```
a = [1,2,3,4,5]
b = reversed(a)    # Return a Reversed object
print(*b)

a = [1,2,3,4,5]
a.reverse()
print(a) # _____________
```

```
# Try
a = [1,2,3,4,5]
b = a.reverse()
print(b) # _____________
print(a) # _____________

# For tuple:
a = (1,2,3,4,5)
b = reversed(a)
print(*b)

# a = (1,2,3,4,5)
# a.reverse() # error: tuple doesn't have the reverse() method.
```

==========

List - Other List specific methods

append, clear, copy, extend,
insert, pop, remove, reverse, sort

*** copy returns the new resultant list, others mutate the original list**

==========

```
# .copy()
a = [1,2,3,4,5]
a1= a.copy() # Return a copy
a1.reverse() # Change contents, return None

b = (1,2,3,4,5)
b1 = b[::-1]    # solution for tuple
```

 # .sort() (different from sorted)

```
a = [5,1,3,8,9,4,2,7,6,10]
b = sorted(a) # Return a new list
print(a)
print(b)

a = [5,1,3,8,9,4,2,7,6,10]
b = a.sort() # Change a's contents, return None
print(a)
print(b)
```

```
#      .extend(), .append(), .insert()
a = b = [*range(5)]
print(a.extend(b))
print(a.append('stop'))
print(a.insert(0,'start'))
print(a)
```

_____

```
#      .pop(), .remove(), .clear(),    del
a = [0,1,2,3,4,5,6,7,8,9]
print(a)
del a[::2]
print(a)
print(a.pop()) # Remove the last item and return it
print(a.remove(5)) # Remove the item and return None
print(a)

a = [0,1,2,3,4,5,6,7,8,9]
print(a.clear())
del a[0:4]
print(a.pop()) # error: pop from empty list
# print(a.remove(5)) # error: x not in list
```

==========

Additional

==========

- from math import **Isqrt**
  return the integer part of the square root of the input.

- **collections.abc** : abstract base classes for containers, e.g. **iterable**
  e.g. Use isinstance and Iterable to check whether an argument is iterable.

- Write a multi-purpose **average** function

  We can write an average function as follows:

```
def average(x):        # [1,2,3]
    return sum(x) / len(x)

a = average([1,2,3]) # gives 2
```

  We can also write the average function as follows:

```
def average(*args): # 1,2,3
    return sum(args) / len(args)

b = average(1,2,3)    # gives 2
```

  We can also write one single average function for both 1,2,3 and [1,2,3] ?

  Note: **max** can work for:

```
c = max(1,2,3) # gives 3
d = max([1,2,3]) # gives 3
```

  Answer:

```
from collections.abc import Iterable
def average(*args):
    if isinstance(args[0], Iterable):
        x = args[0]
    else:
        x = args
    return sum(x)/len(x)

a1 = average([1,2,3]) # gives 2
b1 = average(1,2,3) # gives 2
```

  For the complete solution, see Exercise 1 in https://ccha23.github.io/cs1302i25b/sequence-types/
  (Canvas => CS1302 => Home => Courseware => Jupyter book for CS1302

- Improve isPrime using isqrt and comprehension

  Version 1

```
def isPrime(x):
    for d in range(2,x):
        if x%d == 0:
            return False #stop the function
    return True

print(isPrime(123)) # False
print(isPrime(193)) # True
```

  Version 2

```
from math import isqrt

def isPrime(x):
    for d in range(2,isqrt(x)+1):
        if x%d == 0:
            return False #stop the function
    return True
```

  Explanation 1: for 123
  ==========
  Test if 123 is x*y
  try 2 * __ = 123?
  try 3 * __ = 123? <== found that 3 can divide 123, can stop
  try 4 * __ = 123?
  ..
  ..

  Explanation 2: for 193
  ==========
  square-root of 193 is 13.something

  Test if 193 is any **smaller_factor * larger_factor**

  try 2 * .. = 193?
  try 3 * .. = 193?
  try 4 * .. = 193?<=== Note 3) need to test a even factor? No.

  ..
  try 13 * .. = 193? <=== Note 2) can stop now !!
  ..
  try 192 * .. = 193? <=== Note 1) actually, no need to test until 192

  Version 3 (Sequence_Types.ipynb

```
from math import isqrt
def isPrime(x):
    return all(x%d!=0 for d in range(2, isqrt(x)+1))

def isPrime(x):
    return all(x%d for d in range(2, isqrt(x)+1))

print(isPrime(123))
print(isPrime(193))
```

- **prime_sequence** (Sequence_Types.ipynb): use 2 comprehensions, each create a generator

```
from math import isqrt

def prime_sequence(stop):
    return (x for x in range(2, stop) if
            all(x % d for d in range(2, isqrt(x) + 1)))

print(*prime_sequence(100))
```

- **Filter** and **Map** (Sequence_Types.ipynb)

```
def prime_sequence_(stop):
    return filter(
        lambda x: all(
            map(lambda d: x % d, range(2, isqrt(x) + 1))
        ),
        range(2, stop)
    )

print(*prime_sequence(100))
```

- **composite_sequence** (Sequence_Types.ipynb)

```
def composite_sequence(stop):
    return (x for x in range(2, stop) if any(x % d == 0 for d in range(2, x)))

print(*composite_sequence(100))
```

Another implementation:

```
def sieve_composite_sequence(stop):
    is_composite = [False] * stop    # initialization
    for factor in range(2, stop):
        if is_composite[factor]:
            continue
        for multiple in range(factor ** 2, stop, factor):
            is_composite[multiple] = True
    return (x for x in range(4, stop) if is_composite[x])

result = sieve_composite_sequence(50)
print(*result)
```

Understanding:

```
def sieve_composite_sequence(stop):
    is_composite = [False] * stop    # initialization
    for factor in range(2, stop):
        if is_composite[factor]:
            continue
        for multiple in range(factor ** 2, stop, factor):
            is_composite[multiple] = True
        step_result = (x for x in range(4, stop) if is_composite[x])
        print(factor, ":", *step_result)
    return (x for x in range(4, stop) if is_composite[x])

result = sieve_composite_sequence(50)
print(*result)
```

- **prime_list** (Sequence_Types.ipynb)

```
def prime_list(stop):
    return [
        x for x in range(2, stop) if all([x % d for d in range(2, isqrt(x) + 1)])
    ]    # Enclose comprehension by square brackets

print(prime_list(100))
```

==========

See additional contents and exercises from **Sequence_Types.ipynb**