



## **09 – Tuples**

COMP 125 Programming with Python

# Recording Disclaimer



The synchronous sessions are recorded (audiovisual recordings). The students are not required to keep their cameras on during class.

The audiovisual recordings, presentations, readings and any other works offered as the course materials aim to support remote and online learning. They are only for the personal use of the students. Further use of course materials other than the personal and educational purposes as defined in this disclaimer, such as making copies, reproductions, replications, submission and sharing on different platforms including the digital ones or commercial usages are strictly prohibited and illegal.

The persons violating the above-mentioned prohibitions can be subject to the administrative, civil, and criminal sanctions under the Law on Higher Education Nr. 2547, the By-Law on Disciplinary Matters of Higher Education Students, the Law on Intellectual Property Nr. 5846, the Criminal Law Nr. 5237, the Law on Obligations Nr. 6098, and any other relevant legislation.

The academic expressions, views, and discussions in the course materials including the audio-visual recordings fall within the scope of the freedom of science and art.

# Mutability

- A **mutable** object can be changed after it is created (*e.g., lists*)
- An **immutable** object cannot be (*e.g., strings*)

```
lst = [1, 2, 3]  
lst[0] = 'a'  
lst → ['a', 2, 3]
```

```
s = '123'  
s[0] = 'a'
```



**Error**

# Tuples

- *“Bundles of small amount of data”*: An immutable data type for storing values in an ordered linear collection
- In other words, **immutable lists**
- Use **parentheses ( )** instead of **brackets [ ]** to define them

```
('a', 'b', 'c')
```

```
('number', 1)
```

```
('simba', 'lion', 25)
```

# Tuples vs. Lists

## Tuples

- `len()`, `print()`
- slicing, indexing
- for loops
- `in`
- concatenation
- store arbitrary elements
- **immutable**
  - Can't add, remove or modify elements

## Lists

- `len()`, `print()`
- slicing, indexing
- for loops
- `in`
- concatenation
- store arbitrary elements
- **mutable**
  - `append()`
  - `pop()`
  - assign with indexing
  - `del item`

# Tuples don't support item assignment!

```
tup = ('apple', 0.79, 'WA')
```

```
a = tup[0]    You CAN use index to view the items
```

```
a → 'apple'
```

```
# Let's try to assign a different value to tup[2]
```

```
tup[2] = 'CA'
```



**You CANNOT use index to set one of the items**

# No parentheses needed

- You do not need parentheses to create tuples

```
tuple1 = (True, -3.4, 'hello', 1)
```

```
tuple2 = True, -3.4, 'hello', 1
```

```
print(tuple1)
```

```
print(tuple2)
```

Output:

```
(True, -3.4, 'hello', 1) <class 'tuple'>
```

```
(True, -3.4, 'hello', 1) <class 'tuple'>
```

- However, we **strongly recommend you** to do this for better **readability**

# Tuples

- Empty tuple:

```
tup1 = ()
```

- Tuple including a single value:

```
tup2 = (50, )
```

- **Comma is important!**

```
# check the data types of v1 and v2
```

```
v1 = (50)
```

```
v2 = (50, )
```

```
print(type(v1))
```

→ int

```
print(type(v2))
```

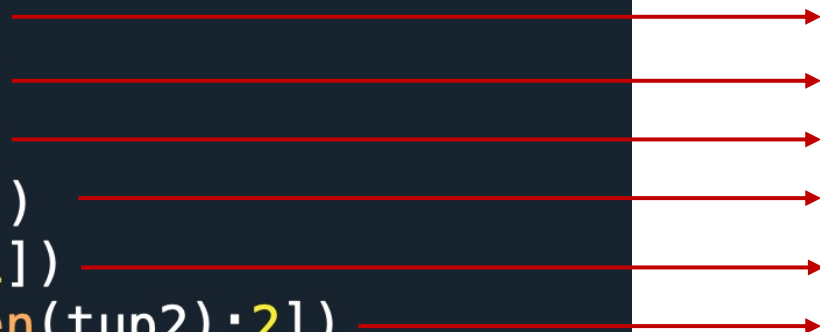
→ tuple



# Tuples

```
tup1 = ('math', 'chem', 2020, 2021)
tup2 = (1, 2, 3, 4, 5, 6, 7)
```

```
print(tup1[0])
print(tup1[-1])
print(tup1[1:])
print(tup2[1:5])
print(tup2[::-1])
print(tup2[1:len(tup2):2])
```



```
math
2021
('chem', 2020, 2021)
(2, 3, 4, 5)
(7, 6, 5, 4, 3, 2, 1)
(2, 4, 6)
```

Quick question: what will be the data type for:

- tup1[0]
- tup1[-1]
- tup2[1:5]

```
print(type(tup1[0]))
print(type(tup1[-1]))
print(type(tup2[1:5]))
```

```
<class 'str'>
<class 'int'>
<class 'tuple'>
```

# Packing and unpacking

- **Packing:** Assigning multiple values (or an iterable) to a tuple

- The *no parentheses* example

```
tup = 'apple', 7.99, 'market'
```

- **Unpacking:** Assigning the contents of the tuple to multiple values

```
product, price, seller = tup
```

```
product → 'apple'
```

- **Other iterables can also be unpacked!**

# Unpacking with tuples

```
a, b, c = 1, 2, 3  
a, b, c = (1, 2, 3)
```

a → 1	b → 2	c → 3
a → 1	b → 2	c → 3

This does not work when the length of the left-hand side **is not equal** to the length of the right-hand side

```
a, b, c = (1, 2)  
a, b = (1, 2, 3)
```



**ValueError**

# Unpacking with other iterables

```
a, b, c = [1, 2, 3]
```

```
a, b, c = '123'
```

```
a, b, c = range(3)
```

```
a → 1
```

```
b → 2
```

```
c → 3
```

```
a → '1'
```

```
b → '2'
```

```
c → '3'
```

```
a → 0
```

```
b → 1
```

```
c → 2
```

This does not work when the length of the left-hand side **is not equal** to the length of the right-hand side

```
a, b, c, d = [1, 2, 3]
```

```
a, b = [1, 2, 3]
```



**ValueError**

# Swapping

- Swapping is commonly needed for programming.  
Python's unpacking feature makes this very elegant!

```
a, b = 1, 2  
a, b = b, a
```

a	→	1	b	→	2
a	→	2	b	→	1

# Tuples vs lists

- Tuples:

- Fixed number of elements, need to know ahead of time!
- Usually used to store small number of elements, that will not be modified in the program

- Lists:

- Unbounded (memory permitting) number of elements
- Preferred to store large number of elements

- Advantages of using tuples over lists

- Processing tuples is faster than processing lists
- Tuples are safe (immutable)

- Tuples and lists can be converted to each other

- `list()` function: converts a tuple to a list
- `tuple()` function: converts a list to a tuple

# Tuple operations: Similar to lists

Python Expression	Result	Description
<code>len( (1, 2, 3) )</code>	3	Length
<code>(1, 2, 3) + (4, 5, 6)</code>	(1, 2, 3, 4, 5, 6)	Concatenation
<code>(1, 2) * 4</code>	(1, 2, 1, 2, 1, 2, 1, 2)	Repetition
<code>3 in (1, 2, 3)</code>	True	Membership
<code>for x in (1, 2, 3):     print(x)</code>	1 2 3	Iteration

# Tuple operations: Unlike lists

- Tuples **do not support** these methods
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`



# Example

- Let's print out the names of the months and number of days

```
names=('January', 'February', 'March', 'April', 'May', 'June', \
      'July', 'August', 'September', 'October', 'November', 'December')
days=(31,28,31,30,31,30,31,31,30,31,30,31)

for i in range(12):
    if i !=1:
        print(f"{i+1:2}. month of the year is {names[i]:9} \
              and has {days[i]} days")
    else:
        print(f"{i+1:2}. month of the year is {names[i]:9} \
              and has {days[i]} days if it is not a leap year")
```

```
1. month of the year is January           and has 31 days
2. month of the year is February          and has 28 days if it is not a leap year
3. month of the year is March             and has 31 days
4. month of the year is April             and has 30 days
5. month of the year is May               and has 31 days
6. month of the year is June              and has 30 days
7. month of the year is July              and has 31 days
8. month of the year is August            and has 31 days
9. month of the year is September         and has 30 days
10. month of the year is October          and has 31 days
11. month of the year is November         and has 30 days
12. month of the year is December        and has 31 days
```

# Tuples & Lists can be combined

- Similar to list-of-lists, one can create:
  - tuple-of-tuples
  - list-of-tuples
  - tuple-of-lists

# Example

- List of students can be mutable, so it can be stored as a list
- But for any given student, attributes should be immutable, hence it can be stored as a tuple

```
def display(students):  
    header=f"{'Name':10} {'Lastname':10} {'Year of birth':15}"  
    print(header)  
    print('- '*len(header))  
  
    for item in students:  
        name, lastname, birthyear = item  
        print(f"{name:10} {lastname:10} {birthyear:<15}")  
    print()  
  
students=[("Barry", "Tone", 2001),("Ravi", "O'Leigh", 1999),\  
          ("Eric", "Widget", 1998),("Desmond", "Eagle", 1999)]  
  
display(students)
```

Name	Lastname	Year of birth
Barry	Tone	2001
Ravi	O'Leigh	1999
Eric	Widget	1998
Desmond	Eagle	1999

# Example

- Suppose that you want to store the 2D Cartesian coordinates of a collection of points. Assume that a new point can be added to this collection, but once it is added, it will not be changed during the program execution.
- What is the best way of keeping these points in Python?

## List of 2D tuples

```
points = [(3, 4), (2, 7), (1, 4)]
```

```
print('before: ', points)
```

```
points.append((8, 9))
```

```
print('after: ', points)
```

```
before: [(3, 4), (2, 7), (1, 4)]  
after:  [(3, 4), (2, 7), (1, 4), (8, 9)]
```

# Example

- Write a function that calculates and returns the sum of the x- and y-coordinates of all points in a list.

```
def coordinate_sum(L):  
    xsum = 0  
    ysum = 0  
    for tup in L:  
        xsum += tup[0]  
        ysum += tup[1]  
    return xsum, ysum  
  
def main():  
    points = [(3, 4), (2, 7), (1, 4), (8, 9)]  
    xsum, ysum = coordinate_sum(points)  
    print('xsum:', xsum, '\tysum:', ysum)  
  
main()
```

```
In [1]: runfile('/Use  
closest.py', wdir='/U  
xsum: 14    ysum: 24
```



# Example

- Now call your function to calculate the centroid of the points, which is the average of the points' coordinates.

```
def calculate_centroid(L):  
    xsum, ysum = coordinate_sum(L)  
    if len(L) > 0:  
        return xsum/len(L), ysum/len(L)  
    return 0, 0  
  
def main():  
    points = [(3, 4), (2, 7), (1, 4), (8, 9)]  
    centroid = calculate_centroid(points)  
    print('centroid:', centroid)  
    print(type(centroid))  
  
main()
```

```
In [1]: runfile('/User  
closest.py', wdir='/Us  
centroid: (3.5, 6.0)  
<class 'tuple'>
```

# Example

- Write another function that takes a list  $L$  of these points together with another point  $P$ . This function should return the data point in  $L$  that is the closest to  $P$  together with this closest distance.

```

import math

def calculate_distance(P1, P2):
    d = math.sqrt((P1[0] - P2[0])**2 + (P1[1] - P2[1])**2)
    return d

def find_closest_point(L, P):
    if len(L) == 0:
        return (), -1.0

    closest_index = -1
    for i in range(len(L)):
        d = calculate_distance(L[i], P)
        if closest_index == -1 or d < closest_distance:
            closest_index = i
            closest_distance = d
    return L[closest_index], closest_distance

def main():
    points = [(3, 4), (2, 7), (1, 4), (8, 9)]
    P = (6, 2)
    closestP, closestD = find_closest_point(points, P)
    print('Closest point: ', closestP)
    print(f'Closest distance: {closestD:0.3}')

main()

```

```

In [1]: runfile('/Users/...
closest.py', wdir='/Users...
Closest point: (3, 4)
Closest distance: 3.61

```



# Sorting a list of tuples

```
lst = [('mango', 3), ('apple', 6), ('lychee', 1), ('apricot', 10)]  
print(sorted(lst))
```

Output:

```
[('apple', 6), ('apricot', 10), ('lychee', 1), ('mango', 3)]
```

**Sorts by the first element in each tuple.**

# Sorting a list of tuples

What about...

```
lst = [(5, 3), ('apple', 6), (3, 1), ('apricot', 10)]  
print(sorted(lst))
```

*TypeError: '<' not supported between instances of 'str' and 'int'*

```
lst = [(False, 3), ('apple', 6), (True, 1), ('apricot', 10)]  
print(sorted(lst))
```

*TypeError: '<' not supported between instances of 'str' and 'bool'*

```
lst = [('False', 3), ('apple', 6), ('True', 1), ('apricot', 10)]  
print(sorted(lst))
```

*Uppercase letters have smaller ASCII values than lowercase letters*

Output:

```
[('False', 3), ('True', 1), ('apple', 6), ('apricot', 10)]
```

# Sorting tuples

```
lst = [3,1,8,5,2,4]
```

```
lst.sort()
```

```
print(lst) → [1, 2, 3, 4, 5, 8]
```

```
lst = [3,1,8,5,2,4]
```

```
sorted_lst = sorted(lst)
```

```
print(sorted_lst) → [1, 2, 3, 4, 5, 8]
```

```
tup = (3,1,8,5,2,4)
```

```
tup.sort()
```

 **AttributeError**

```
sorted_tup = sorted(tup)
```

```
print(sorted_tup) → [1, 2, 3, 4, 5, 8]
```

 **<class 'list'>**

# Sorting strings

- You can also use the **sorted** function with strings

```
print(sorted('anaconda'))
```

```
['a', 'a', 'a', 'c', 'd', 'n', 'n', 'o'] <class 'list'>
```

# Let's practice

Which of the following statements creates a tuple?

a. `values = [1, 2, 3, 4]`

b. `values = {1, 2, 3, 4}`

c. `values = (1)`

d. `values = (1,)`

# Let's practice

- Modify the first item, 22, of a list inside a following tuple to 99

```
tuple1 = (11, [22, 33], 44, 55)
```

```
tuple1 = (11, [22, 33], 44, 55)
```

```
tuple1[1][0] = 99
```

```
print(tuple1)
```



**Can we do this?**

**YES, WE CAN!**

**We are changing an item  
inside a LIST.**

# Packing lists: zip()

```
mnames = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',  
          'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']  
mdays = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]  
  
# pack two lists  
pairs = list(zip(mnames, mdays))  
print('Pairs:\n', pairs)  
  
print()  
for name, days in pairs:  
    print(f'{name}:{days}')
```

```
Pairs:  
[('Jan', 31), ('Feb', 28), ('Mar', 31),  
 ('Apr', 30), ('May', 31), ('Jun', 30),  
 ('Jul', 31), ('Aug', 31), ('Sep', 30),  
 ('Oct', 31), ('Nov', 30), ('Dec', 31)]  
  
Jan:31  
Feb:28  
Mar:31  
Apr:30  
May:31  
Jun:30  
Jul:31  
Aug:31  
Sep:30  
Oct:31  
Nov:30  
Dec:31
```