

01076105, 01076106

Object Oriented Programming

Object Oriented Programming Project

**From C to Python #2**

# List

- **List** เป็นโครงสร้างข้อมูลของ Python ที่คล้าย Array แต่สามารถเก็บข้อมูลต่างชนิดกันได้ด้วย โดย List จะใช้เครื่องหมาย [ ] ในการกำหนดขอบเขตของ List

```
# Creating a List
List = []
print("Blank List: ")
print(List)

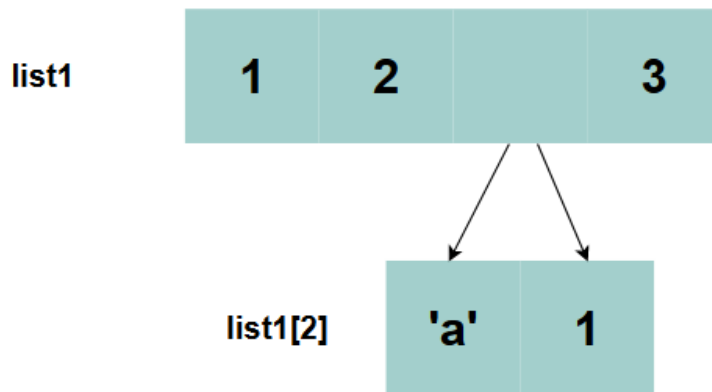
# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)
```

```
# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Computer', 4, 'Eng', 6, 'KMITL']
print("\nList with the use of Mixed Values: ")
print(List)
```

# List

- นอกจากนั้นข้างใน List อาจจะมี List อื่นๆ อีกก็ได้ เรียกว่า Nested List

```
# creating list  
nested_list = [1, 2, ['a', 1], 3]
```



- การอ้างถึงข้อมูลแต่ละตัว สามารถทำได้โดยการอ้าง Index โดยข้อมูลตัวแรก จะอยู่ที่ Index 0 เช่น จากโปรแกรมข้างต้น
  - `nested_list[0]` จะหมายถึง 1
  - `nested_list[2]` จะหมายถึง ['a', 1] และ `nested_list[2][1]` จะหมายถึง 1

# List


- Index สามารถใช้เป็นลบได้ด้วย โดย -1 จะหมายถึง สมาชิกตัวสุดท้ายของ List และ -2 หมายถึงสมาชิกตัวรองสุดท้ายของ List พุดง่ายๆ คือ การนับจากหลังมาข้างหน้า

```
languages = ["Python", "Swift", "C++"]  
  
# access item at index 0  
print(languages[-1])    # C++  
  
# access item at index 2  
print(languages[-3])    # Python
```

	"Python"	"Swift"	"C++"
index →	0	1	2
negative index →	-3	-2	-1

# List

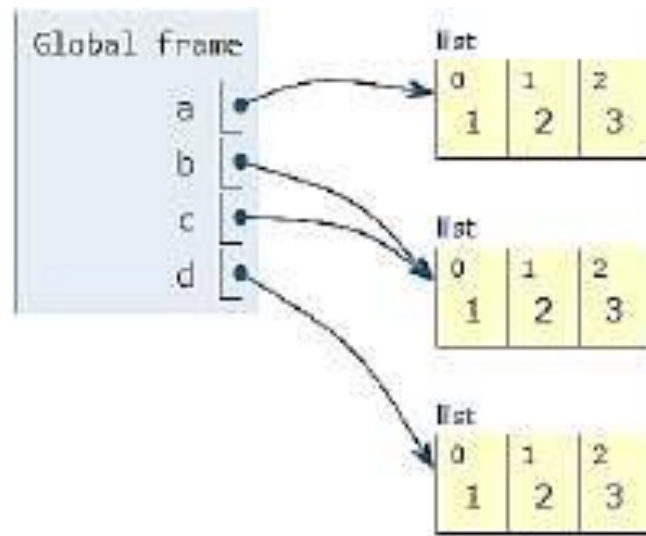
- List สามารถใช้ Slicing ได้เช่นเดียวกับ String

```
main.py ×  Console Shell  
1 list1 = ['physics', 'chemistry',  
  'calculus', 'biology'];  
2 list2 = [1, 2, 3, 4, 5, 6, 7];  
3 print("list1[0]:", list1[0])  
4 print("list1[-1]:", list1[-1])  
5 print("list2[3]:", list2[3])  
6 print("list2[-4]:", list2[-4])  
7 print("list2[1:5]:", list2[1:5])  
8 print("list2[:2]:", list2[:2])  
9 print("list2[2::2]:",  
  list2[2::2])  
10 print("list2[2:7:2]:",  
  list2[2:7:2])  
11 print("list2[:7]:", list2[:7])  
12 print("list2[4:]:", list2[4:])
```

```
list1[0]: physics  
list1[-1]: biology  
list2[3]: 4  
list2[-4]: 4  
list2[1:5]: [2, 3, 4, 5]  
list2[:2]: [1, 3, 5, 7]  
list2[2::2]: [3, 5, 7]  
list2[2:7:2]: [3, 5, 7]  
list2[:7]: [1, 2, 3, 4, 5, 6, 7]  
list2[4:]: [5, 6, 7]  
➤
```

# List

- คำสั่ง list ใช้ในการสร้าง list
- `x = list('abcde')` จะได้ `x = ['a', 'b', 'c', 'd', 'e']`
- `a = [1,2,3]; b = [1,2,3]; c = b; d = list(b)` หลังจากทำคำสั่งจะได้



# List

- การเพิ่มข้อมูลเข้าไปใน List สามารถใช้ได้หลายวิธี
- โดยการ +

```
main.py x
1 flowers = ['Rose', 'Lily', 'Tulip']
2 flowers += ['Jasmine']
3
```

```
Console Shell
['Rose', 'Lily', 'Tulip', 'Jasmine']
>
```

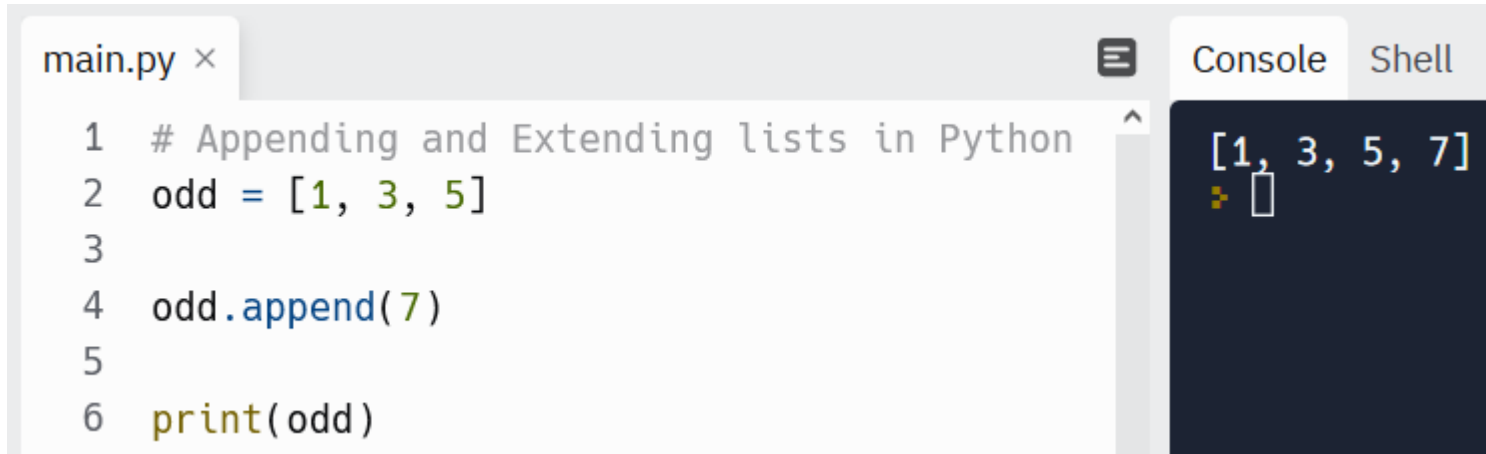
- แต่หากเขียนโปรแกรมแบบนี้ ผลจะเป็นอีกแบบหนึ่ง

```
main.py x
1 flowers = ['Rose', 'Lily', 'Tulip']
2 flowers += 'Jasmine'
3
```

```
['Rose', 'Lily', 'Tulip', 'J', 'a', 's', 'm', 'i', 'n', 'e']
>
```

# List

- เพราะ List เป็น Object จึงสามารถเพิ่มโดยใช้ method append ได้

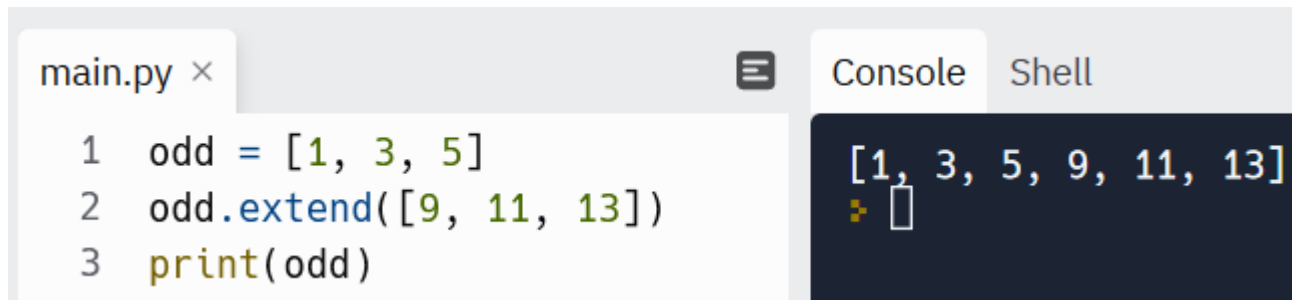


```
main.py ×
1 # Appending and Extending lists in Python
2 odd = [1, 3, 5]
3
4 odd.append(7)
5
6 print(odd)
```

Console Shell

```
[1, 3, 5, 7]
```

- การเพิ่มโดยใช้ extend (ข้อมูลที่ extend ต้องเป็น list เช่นกัน)



```
main.py ×
1 odd = [1, 3, 5]
2 odd.extend([9, 11, 13])
3 print(odd)
```

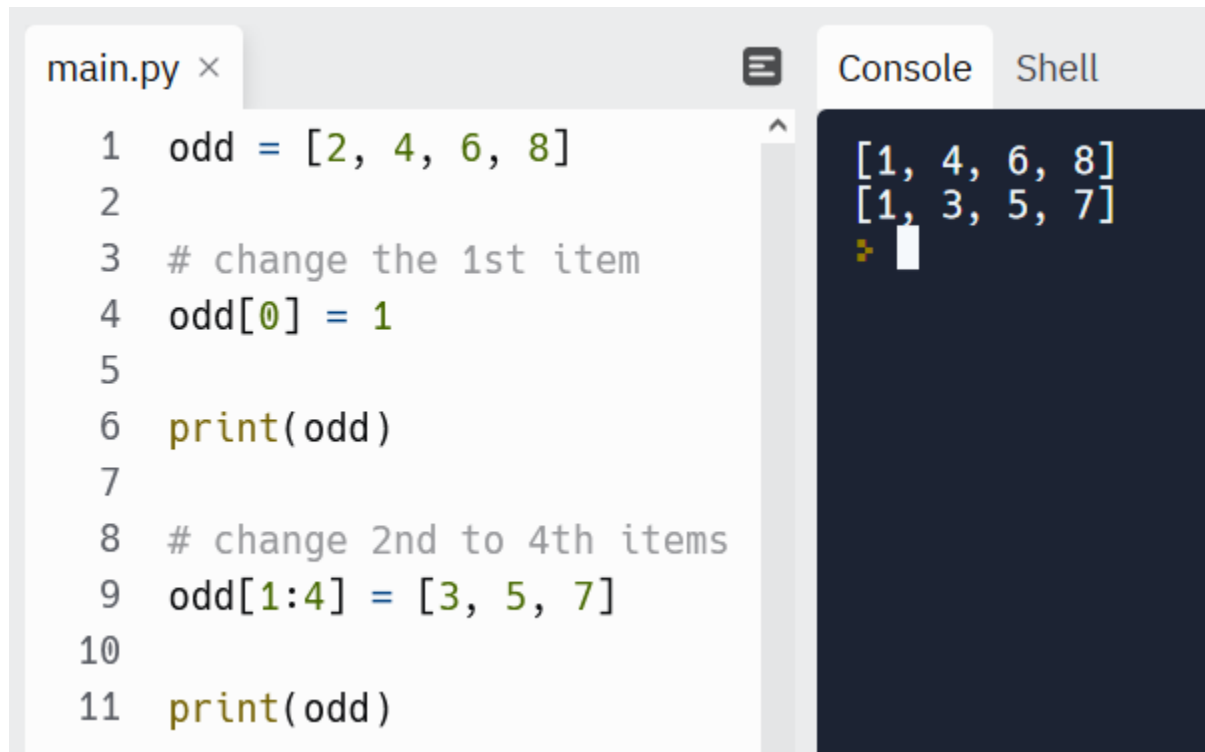
Console Shell

```
[1, 3, 5, 9, 11, 13]
```



# List

- การแก้ไขข้อมูลใน List จะเป็นการกำหนดค่าใหม่ให้กับแต่ละตำแหน่งที่อยู่ใน List โดยตรง



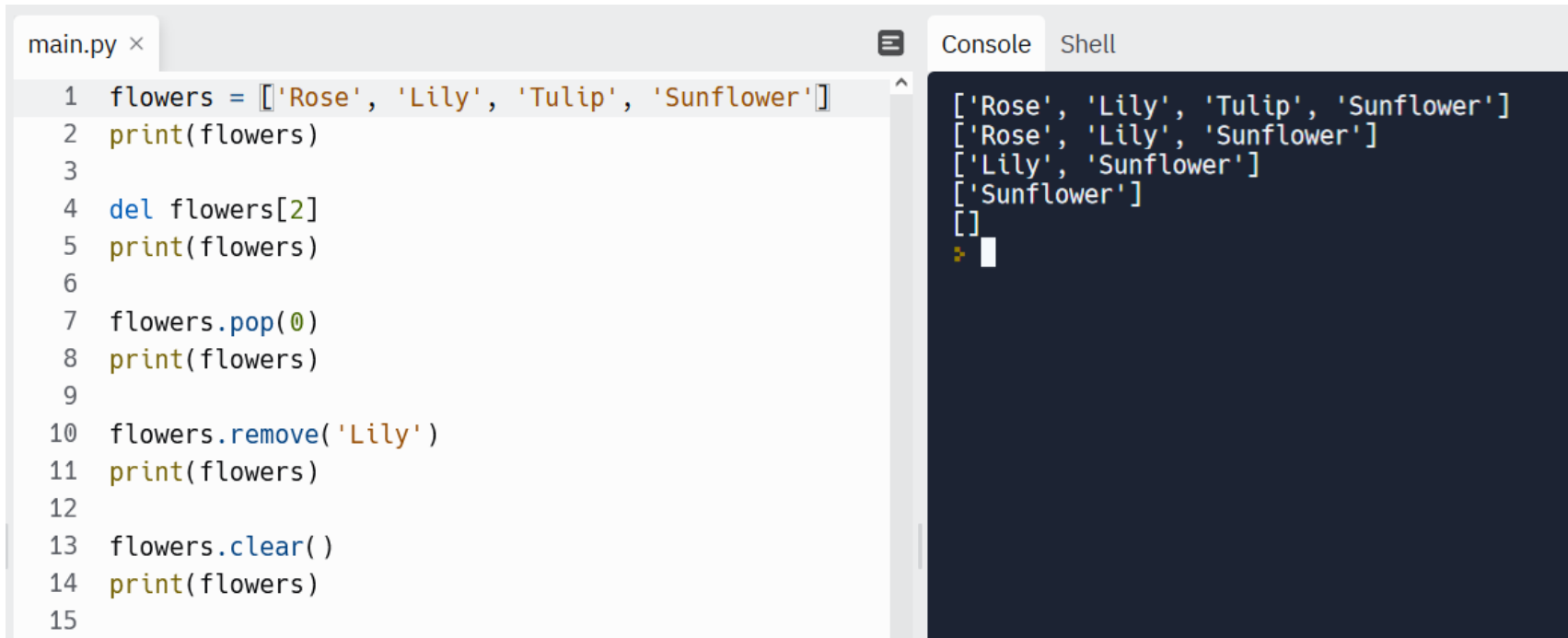
```
main.py ×
1 odd = [2, 4, 6, 8]
2
3 # change the 1st item
4 odd[0] = 1
5
6 print(odd)
7
8 # change 2nd to 4th items
9 odd[1:4] = [3, 5, 7]
10
11 print(odd)
```

Console Shell

```
[1, 4, 6, 8]
[1, 3, 5, 7]
```

# List

- สำหรับการลบข้อมูลจาก List สามารถทำได้หลายวิธีเช่นกัน



The screenshot shows a Python IDE with a file named `main.py`. The code in the editor performs the following operations on a list named `flowers`:

```
1 flowers = ['Rose', 'Lily', 'Tulip', 'Sunflower']
2 print(flowers)
3
4 del flowers[2]
5 print(flowers)
6
7 flowers.pop(0)
8 print(flowers)
9
10 flowers.remove('Lily')
11 print(flowers)
12
13 flowers.clear()
14 print(flowers)
15
```

The `Console` tab on the right displays the output of these operations:

```
['Rose', 'Lily', 'Tulip', 'Sunflower']
['Rose', 'Lily', 'Sunflower']
['Lily', 'Sunflower']
['Sunflower']
[]
>
```

- คำสั่ง `del` จะต้องระบุตำแหน่ง (ใช้ slicing ได้) , คำสั่ง `pop` จะคล้ายกับ `del` แต่สามารถ `return` ตัวที่ลบออกมาได้ (ถ้าไม่ระบุจะได้ตัวสุดท้าย), คำสั่ง `remove` จะต้องระบุข้อมูล

# List

- ฟังก์ชันที่เกี่ยวข้องกับ List อื่นๆ
  - `x.sort()` ทำให้ข้อมูลในลิสต์ `x` เรียงจากน้อยไปมาก คำสั่งนี้ไม่มี return คืนกลับมา คือ เป็นการเรียงใน List ต้นฉบับ
  - `sorted(x)` คืนลิสต์ที่มีค่าเหมือนกับใน `x` แต่เรียงลำดับข้อมูลจากน้อยไปมาก (List ต้นฉบับไม่เปลี่ยนแปลง)
  - `sum(x)` คืนผลรวม (summary) ของตัวเลขที่อยู่ในลิสต์ `x`
  - `max(x)` คืนค่ามากสุดในลิสต์ `x`, `min(x)` คืนค่าน้อยสุดในลิสต์ `x`
  - `x.count(e)` คืนจำนวนครั้งที่ `e` ปรากฏในลิสต์ `x`
  - `sorted`, `sum`, `max` ไม่ใช่ฟังก์ชันของ Object List แต่เป็น Reduce Function

# List

- เราสามารถตรวจสอบได้ว่ามีสมาชิกนั้นอยู่ใน List นั้นหรือไม่ (Membership)

```
main.py ×  
1 my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']  
2  
3 # Output: True  
4 print('p' in my_list)  
5  
6 # Output: False  
7 print('a' in my_list)  
8  
9 # Output: True  
10 print('c' not in my_list)  
11
```

Console Shell

```
True  
False  
True  
>
```

# List

- เราสามารถเปรียบเทียบ List ได้ โดยวิธีการเปรียบเทียบจะเริ่มจากเปรียบเทียบสมาชิกตัวที่ 1 ของ List ทั้งสอง ถ้าสมาชิกตัวแรกมากกว่าหรือน้อยกว่า ก็จะใช้ผลลัพธ์นั้นเป็นคำตอบ แต่ถ้าหากมีค่าเท่ากัน ก็จะเลื่อนไปเปรียบเทียบสมาชิกอันดับถัดไปเรื่อยๆ

main.py ×

```
1 lst1 = [1, 2, 3, 4, 5]
2 lst2 = [9, 8, 7, 6, 5]
3 lst3 = [9, 8, 7, 6, 5]
4 lst4 = [8, 7]
5 print("lst1 < lst2 :", lst1 < lst2)
6 print("lst1 > lst2 :", lst1 > lst2)
7 print("lst2 >= lst1 :", lst2 >= lst1)
8 print("lst2 == lst3 :", lst2 == lst3)
```



Console

Shell

```
lst1 < lst2 : True
lst1 > lst2 : False
lst2 >= lst1 : True
lst2 == lst3 : True
```

# List comprehension

- เป็นวิธีการขั้นสูง ที่ทำให้เขียนโปรแกรมเพื่อสร้าง List ได้สั้นลงในบางกรณี ดูตัวอย่างโปรแกรม

```
main.py x
1 h_letters = []
2
3 for letter in 'human':
4     h_letters.append(letter)
```

Console Shell

```
['h', 'u', 'm', 'a', 'n']
```

- หากเขียนในแบบ List Comprehension จะเขียนได้เป็น

```
main.py x
1 h_letters = [ letter for letter in 'human' ]
2 print( h_letters)
```

Console Shell

```
['h', 'u', 'm', 'a', 'n']
```

- จะเห็นว่าผลการทำงานเหมือนเดิม แต่โปรแกรมสั้นลง

# List comprehension

- รูปแบบของ List Comprehension จะเริ่มด้วย expression แล้วตามด้วย for loop โดย expression จะกระทำกับแต่ละ element ใน list แล้วคืนค่ากลับมา เนื่องจาก list comprehension อยู่ใน list ดังนั้นค่าที่คืนกลับมาก็จะอยู่ในอีก list หนึ่ง

## Syntax of List Comprehension

```
[expression for item in list]
```

[expression for item in list]

[letter for letter in 'human']

# List comprehension

```
for (set of values to iterate):  
    if (conditional filtering):  
        output_expression()
```



```
[ output_expression() for(set of values to iterate) if(conditional filtering) ]
```



# List comprehension

- การใช้งาน List comprehension มีหลายแบบ รูปแบบแรก คือ **Map** โดยเป็นการกระทำกับทุกสมาชิกใน List ในรูปแบบใดรูปแบบหนึ่ง โดยหลังจากทำงาน จะมีจำนวนสมาชิกเท่าเดิม
- ตัวอย่าง เป็นการนำ List เดิมมากำลังสอง

main.py ×

```
1 square = [num**2 for num in range(1,10)]
2 print(square)
```



Console

Shell

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

- ตัวอย่าง เป็นการนำ List เดิมมาเปลี่ยนเป็นตัวใหญ่

main.py ×

```
1 fruits = ["apple", "banana", "cherry",
2          "kiwi", "mango"]
3 newlist = [x.upper() for x in fruits]
4 print(newlist)
```



Console

Shell

```
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
>
```

# List comprehension

- **Map** สามารถเปลี่ยนแปลงเฉพาะ สมาชิกบางตัวตามเงื่อนไขที่กำหนดได้ โดยใช้ if
- โดย if จะใช้ในการเลือกสมาชิกบางตัว ที่ for ส่งกลับมา เพื่อให้มีการกระทำเพิ่มเติมกับสมาชิกตัวนั้น (ข้อมูลยังคงมีจำนวนเท่ากับข้อมูลต้นฉบับ) ดังนั้น if นี้จึงไม่ใช่ conditional filtering แต่เป็น if ที่กระทำกับข้อมูลที่ส่งกลับจาก for
- จากตัวอย่าง เงื่อนไขที่กำหนด คือ เมื่อ  $x = 3$  ให้เปลี่ยนเป็นคำว่า three แต่ตัวอื่นไม่ต้องเปลี่ยน

main.py x

```
1 newlist = ["three" if x == 3 else x \
2           for x in range(1,10)]
3 print(newlist)
```



Console

Shell

```
[1, 2, 'three', 4, 5, 6, 7, 8, 9]
```



# List comprehension

- รูปแบบที่ 2 เรียกว่า **Filter** โดยเป็นการเลือกสมาชิกบางตัวตามเงื่อนไขที่กำหนด ดังนั้นผลลัพธ์จะมีจำนวนสมาชิกลดลง (if แบบนี้เป็น conditional filtering)
- ตัวอย่าง เป็นการสร้าง List ของเลขคู่ จาก List ของเลขตั้งแต่ 1-20

```
main.py x
1 number_list = [ x for x in range(20) \
2                 if x % 2 == 0]
3 print(number_list)
```

Console Shell

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- ตัวอย่างนี้ จะมีหลายเงื่อนไขก็ได้ จะเห็นว่าระหว่าง if จะเหมือนกับมี and เชื่อม

```
main.py x
1 num_list = [y for y in range(100) \
2             if y % 2 == 0 if y % 5 == 0]
3 print(num_list)
```

Console Shell

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

# List comprehension

- สำหรับรูปแบบที่ 3 คือ ใช้ร่วมกันทั้ง **Map** และ **Filter**
- ตัวอย่าง สมมติมี List ของคะแนน ซึ่งหากต้องการจะหาเฉพาะคนที่ได้คะแนนน้อยกว่า 20 และเพิ่มให้อีก 10 คะแนน

```
main.py × +  
1 score = [66, 90, 68, 59, 76, 20, 60, 88, 74, 81, 65, 10]  
2  
3 b = [e+10 for e in score if e <= 20]  
4 print(b)  
5
```

```
>_ Console ×  
[30, 20]  
█
```

# List comprehension

- จะเห็นว่ากรณีที่โปรแกรมไม่ซับซ้อนมากเกินไป สามารถใช้ List Comprehension ได้ แต่ข้อเสีย คือ โปรแกรมอ่านยากขึ้น ดังนั้นถ้าไม่ชำนาญควรเขียนแบบเดิม หรือใช้แต่แบบง่ายๆ

```
main.py x
1 transposed = []
2 matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
3
4 ▼ for i in range(len(matrix[0])):
5     transposed_row = []
6
7 ▼     for row in matrix:
8         transposed_row.append(row[i])
9         transposed.append(transposed_row)
10
11 print(transposed)
```

Console Shell

```
[[1, 4], [2, 5], [3, 6], [4, 8]]
```

```
main.py x
1 matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
2 transpose = [[row[i] for row in matrix]\
3               for i in range(4)]
4 print (transpose)
```

Console Shell

```
[[1, 4], [2, 5], [3, 6], [4, 8]]
```

# List comprehension

- การใช้ List Comprehension ที่น่าสนใจ

```
x = [int(e) for e in input().split()]
```

อ่าน String ด้วย input() และแยกออกเป็น List ของ String ด้วย Split() จากนั้น  
นำ String มาแปลงเป็นจำนวนเต็มและเก็บใน List ประโยชน์ คือ รับข้อมูลที่หลายค่า

```
t = ','.join([str(e) for e in x])
```

สร้างลำดับของผลลัพธ์ที่คั่นด้วยเครื่องหมาย , โดยแปลงตัวเลขใน List เป็น String  
แล้วนำไป join กันอีกที

```
c = sum([1 for e in x if e%2==0])
```

นับว่า List มีจำนวนคู่กี่ตัว โดยสร้าง List ที่เพิ่มเลข 1 ทุกครั้งที่พบจำนวนคู่ใน List แล้ว Sum

```
b = [(1 if x[i] >= 0 else -1) for i in range(len(x))]
```

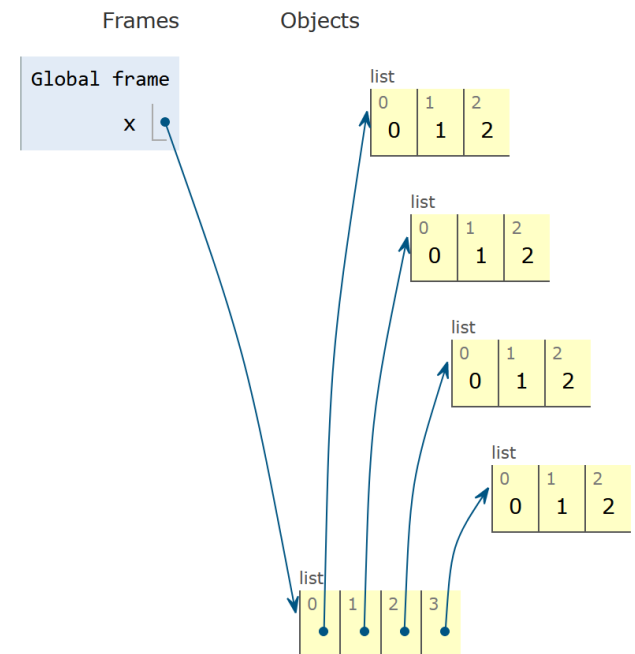
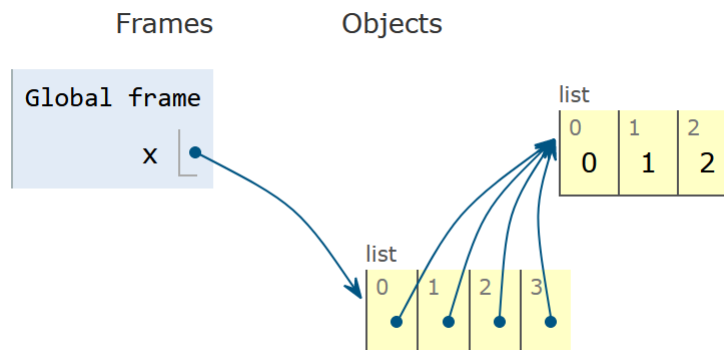
สร้าง List b จาก List x โดยถ้า  $x[i] \geq 0$  ให้เป็น 1 มิฉะนั้นให้เป็น -1

# List comprehension

- คำสั่งในบรรทัดใด ต่อไปนี้ที่ต่างออกไป

```
x = [ [e for e in range(3)] for k in range(4) ]  
x = [ list(range(3)) for k in range(4) ]  
x = [ [0,1,2] for k in range(4) ]  
x = [ [0,1,2] ] *4  
print(x)
```

- แม้ว่าผลลัพธ์จะเหมือนกันก็ตามแต่การทำงานต่างกัน
- บรรทัด 1-3 จะได้รูปขวา แต่บรรทัดที่ 4 จะเป็นรูปล่าง



# List comprehension

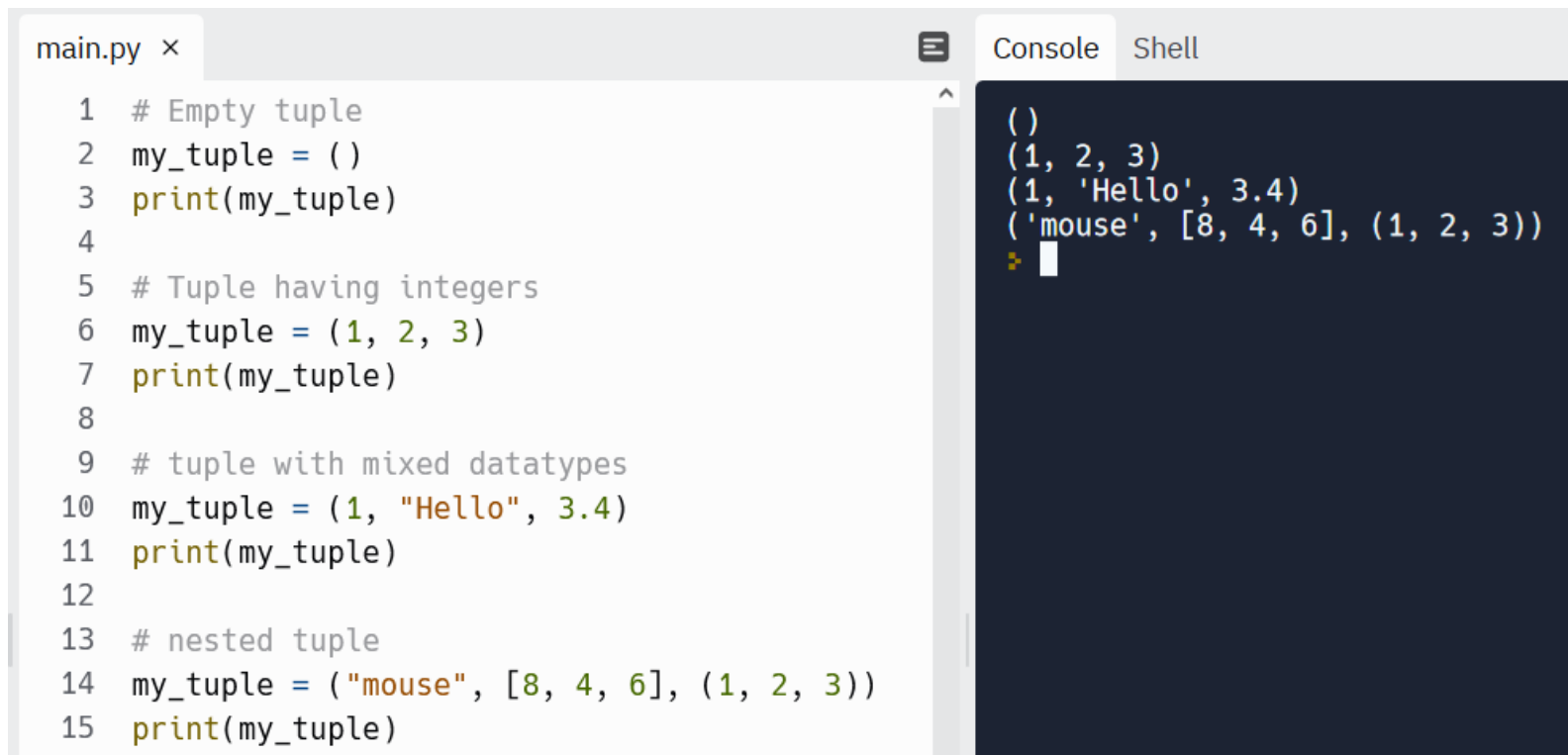
- ตัวอย่าง ต้องการหาตัวเลขอยู่ระหว่าง 1-100 ที่หารด้วยตัวเลขตั้งแต่ 2-9 ลงตัว

```
1  nums = range(1,100)
2  answer = [num for num in nums if True in
3             [True for divisor in range(2,10)
4              if num % divisor == 0]]
5  print(answer)
```



# Tuple

- Tuple เป็นโครงสร้างข้อมูลที่คล้ายกับ List แต่ Tuple เป็นแบบ Immutable (คือเมื่อกำหนดขึ้นแล้วจะแก้ไขไม่ได้ เช่นเดียวกับ String)
- Tuple จะใช้สัญลักษณ์ () ในการกำหนด



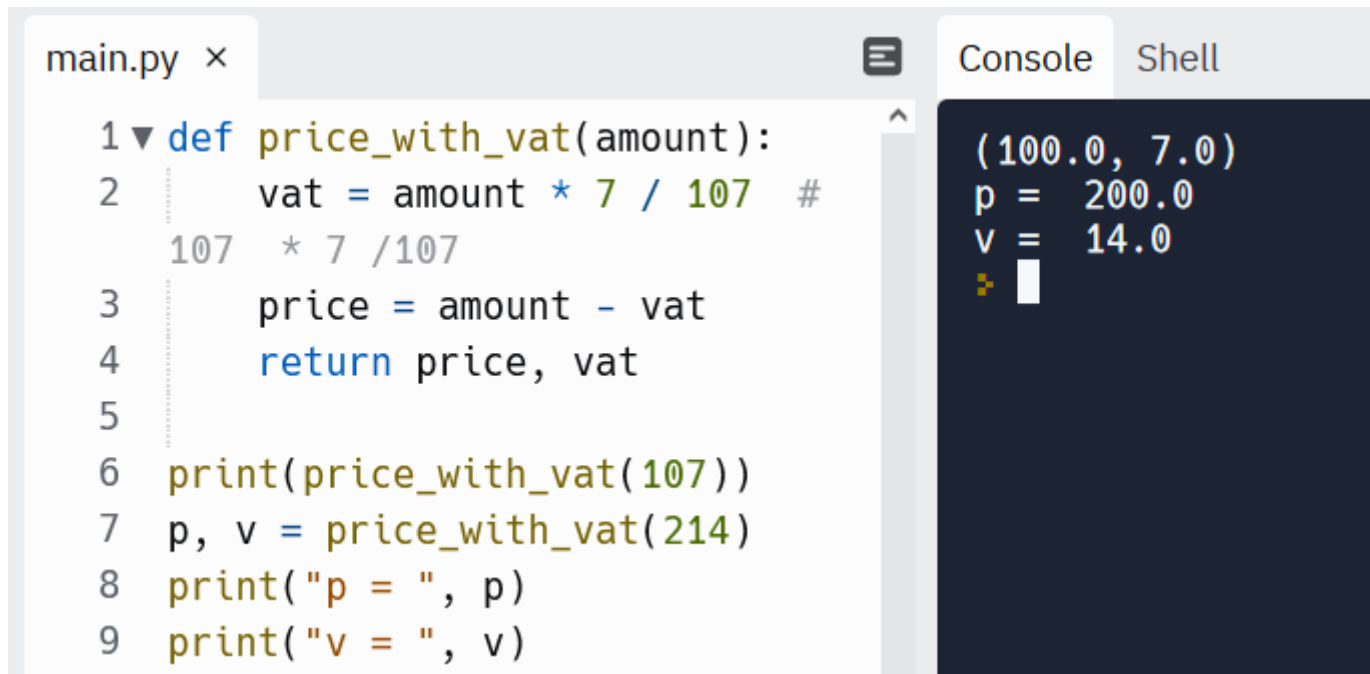
The screenshot shows a Python IDE with a file named 'main.py' and a 'Console' window. The code in 'main.py' demonstrates various ways to create tuples: an empty tuple, a tuple of integers, a tuple with mixed data types, and a nested tuple. The 'Console' window shows the output of these operations, confirming that tuples are immutable and can contain any data type.

```
main.py ×
1 # Empty tuple
2 my_tuple = ()
3 print(my_tuple)
4
5 # Tuple having integers
6 my_tuple = (1, 2, 3)
7 print(my_tuple)
8
9 # tuple with mixed datatypes
10 my_tuple = (1, "Hello", 3.4)
11 print(my_tuple)
12
13 # nested tuple
14 my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
15 print(my_tuple)
```

```
Console
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
>
```

# Tuple

- เนื่องจาก Tuple เป็น Immutable จึงมักใช้งานกับข้อมูลที่ไม่มีการเปลี่ยนแปลง เช่น ข้อมูลที่ return จากฟังก์ชัน กรณีที่มีหลายค่า ก็จะ return ออกมาเป็น Tuple



The screenshot shows a Python IDE with a file named 'main.py' and a 'Console' tab. The code in 'main.py' defines a function 'price\_with\_vat' that takes an 'amount' and returns a tuple of (price, vat). The function calculates the vat as 7% of the amount and subtracts it from the amount to get the price. The console shows the output of the function calls: (100.0, 7.0) for amount 107, and p = 200.0, v = 14.0 for amount 214.

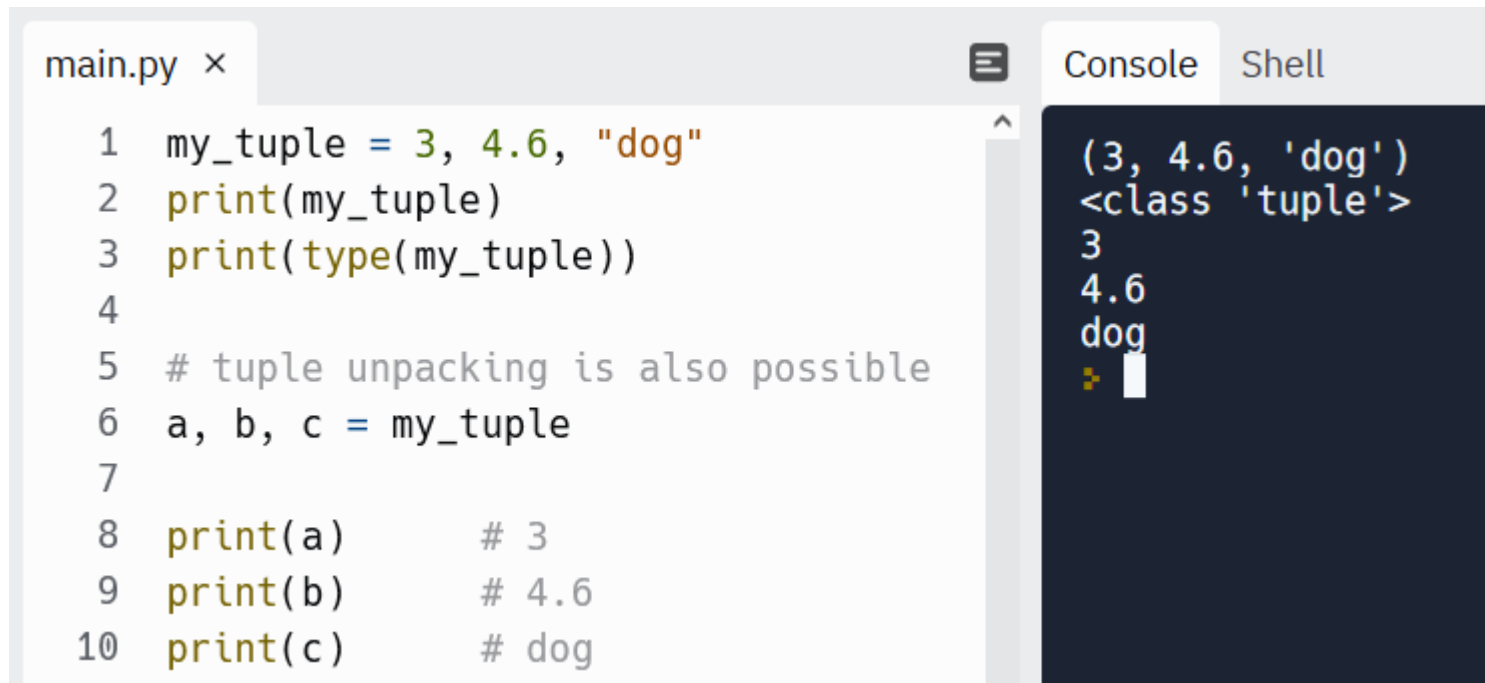
```
main.py ×  
1 ▼ def price_with_vat(amount):  
2     vat = amount * 7 / 107 #  
   107 * 7 / 107  
3     price = amount - vat  
4     return price, vat  
5  
6 print(price_with_vat(107))  
7 p, v = price_with_vat(214)  
8 print("p = ", p)  
9 print("v = ", v)
```

Console  
Shell

```
(100.0, 7.0)  
p = 200.0  
v = 14.0  
➤
```

# Tuple

- การสร้าง Tuple ไม่จำเป็นต้องใช้ ( ) เสมอไป จริงๆ แล้วสิ่งที่สร้าง Tuple คือ เครื่องหมาย , (เรียกว่า Tuple Packing) และยังสามารถ unpack กระจายออกมาที่ตัวแปรอื่นได้ด้วยตามตัวอย่าง



The screenshot shows a Python IDE with a file named 'main.py'. The code in the editor demonstrates tuple creation and unpacking. The console output shows the tuple representation and the unpacked values.

```
main.py ×  
1 my_tuple = 3, 4.6, "dog"  
2 print(my_tuple)  
3 print(type(my_tuple))  
4  
5 # tuple unpacking is also possible  
6 a, b, c = my_tuple  
7  
8 print(a)      # 3  
9 print(b)      # 4.6  
10 print(c)     # dog
```

Console output:

```
(3, 4.6, 'dog')  
<class 'tuple'>  
3  
4.6  
dog
```

# Tuple

- ในคำสั่งแรก แม้จะใส่วงเล็บ แต่ก็ไม่ได้เกิดตัวแปรชนิด Tuple แต่ถ้าใส่เครื่องหมาย , ต่อท้าย ก็จะเป็น Tuple เพราะสิ่งที่สร้าง Tuple คือ , เช่นในคำสั่งที่ 3 ไม่มีวงเล็บ แต่ก็ยังเป็นชนิด Tuple

main.py ×

```
1 my_tuple = ("hello")
2 print(type(my_tuple)) # <class 'str'>
3
4 # Creating a tuple having one element
5 my_tuple = ("hello",)
6 print(type(my_tuple)) # <class 'tuple'>
7
8 # Parentheses is optional
9 my_tuple = "hello",
10 print(type(my_tuple)) # <class 'tuple'>
11
```



Console

Shell

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

# Tuple

- หลายๆ ครั้งที่เราใช้ Tuple โดยไม่ได้ตั้งใจ เช่น เขียนว่า

a, b, c = 10, 20, 30

เป็นการสร้าง Tuple ที่มีสมาชิก 10, 20, 30 จากนั้นจึง Unpack ให้กับตัวแปร a, b, c

- จริงๆ แล้วกลไก unpack สามารถใช้งานได้หลากหลาย

a, b, c = [10, 20, 30]

a, b, c = 'XYZ'

for e in 10, 20, 'abc':

for e in 'abc':

a, b = b, a

# Tuple

- สำหรับการเข้าถึง Tuple จะเหมือนกับ List ตามตัวอย่าง

```
main.py x
1 # Accessing tuple elements using indexing
2 my_tuple = ('p','e','r','m','i','t')
3
4 print(my_tuple[0]) # 'p'
5 print(my_tuple[5]) # 't'
6
7 # nested tuple
8 n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
9
10 # nested index
11 print(n_tuple[0][3]) # 's'
12 print(n_tuple[1][1]) # 4
13
14 # Negative indexing for accessing tuple elements
15 my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
16
17 # Output: 't'
18 print(my_tuple[-1])
19
20 # Output: 'p'
21 print(my_tuple[-6])
```

Console Shell

p  
t  
s  
4  
t  
p

# Tuple

- Tuple สามารถทำ Slicing ได้

```
main.py ×
1 # Accessing tuple elements using slicing
2 my_tuple = ('e','n','g','i','n','e','e','r')
3
4 # elements 2nd to 4th
5 print(my_tuple[1:4])
6
7 # elements beginning to 2nd
8 print(my_tuple[:-7])
9
10 # elements 8th to end
11 print(my_tuple[7:])
12
13 # elements beginning to end
14 print(my_tuple[:])
```

```
Console Shell
('n', 'g', 'i')
('e',)
('r',)
('e', 'n', 'g', 'i', 'n', 'e', 'e', 'r')
```

# Tuple

- การเปลี่ยนแปลงค่าไม่สามารถทำได้ เพราะเป็น Immutable
- **แต่...** การเปลี่ยนแปลงค่าสมาชิกที่เป็น Mutable สามารถเปลี่ยนได้ เพราะที่ไม่เปลี่ยนคือตำแหน่ง (id) ของสมาชิกไม่เปลี่ยน (สำหรับการลบ ใช้ del)

main.py ×

```
1 # Changing tuple values
2 my_tuple = (4, 2, 3, [6, 5])
3
4 # However, item of mutable element can be changed
5 my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
6 print(my_tuple)
7
8 # Tuples can be reassigned
9 my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm')
10
11 print(my_tuple)
12
```

Console

Shell

```
(4, 2, 3, [9, 5])
('p', 'r', 'o', 'g', 'r', 'a', 'm')
>
```



# Tuple

- สำหรับ method ต่างๆ ของ Tuple ก็คล้ายกับ List แต่จะไม่มี method ที่จะไปเปลี่ยนค่าของ Tuple

```
main.py ×
1 my_tuple = ('a', 'p', 'p', 'l', 'e',)
2
3 print(my_tuple.count('p')) # Output: 2
4 print(my_tuple.index('l')) # Output: 3
```

Console Shell

```
2
3
>
```

- สามารถใช้ membership ได้

```
main.py ×
1 my_tuple = ('a', 'p', 'p', 'l', 'e',)
2
3 # In operation
4 print('a' in my_tuple)
5 # Not in operation
6 print('g' not in my_tuple)
```

Console Shell

```
True
True
>
```

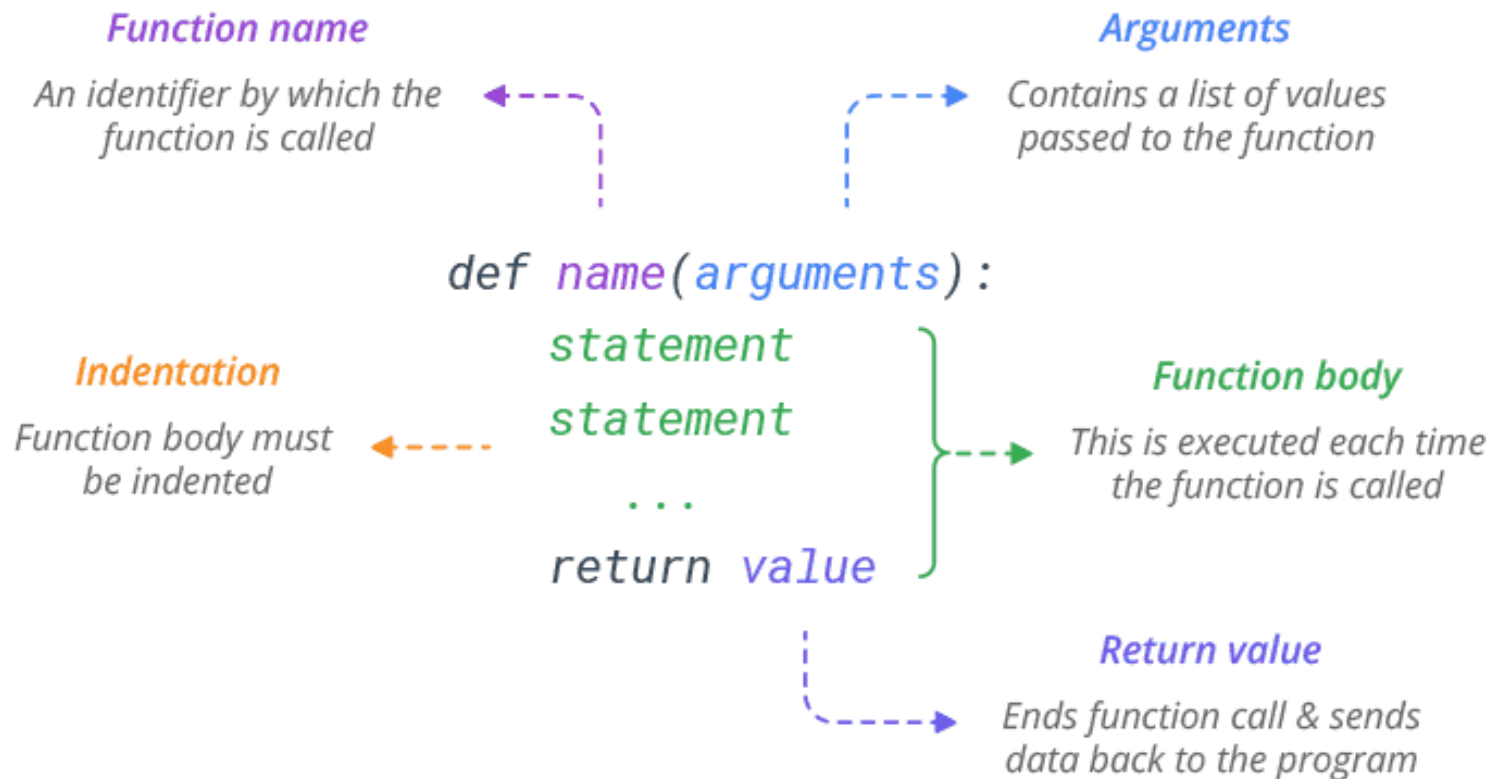
# Function

- จุดเด่นอย่างหนึ่งของภาษา Python คือ มี Function ให้ใช้งานมาก ทั้งที่เป็น Standard และที่มีผู้พัฒนาเพิ่มเติม
- สามารถดูฟังก์ชัน Standard ได้จาก Link ด้านล่าง
- <https://docs.python.org/3/library/functions.html>

Built-in Functions			
<b>A</b> abs() aiter() all() any() anext() ascii()	<b>E</b> enumerate() eval() exec()	<b>L</b> len() list() locals()	<b>R</b> range() repr() reversed() round()
<b>B</b> bin() bool() breakpoint() bytearray() bytes()	<b>F</b> filter() float() format() frozenset()	<b>M</b> map() max() memoryview() min()	<b>S</b> set() setattr() slice() sorted() staticmethod() str() sum() super()
<b>C</b> callable() chr() classmethod() compile() complex()	<b>G</b> getattr() globals()	<b>N</b> next()	<b>T</b> tuple() type()
<b>D</b> delattr() dict() dir() divmod()	<b>H</b> hasattr() hash() help() hex()	<b>O</b> object() oct() open() ord()	<b>V</b> vars()
	<b>I</b> id() input() int() isinstance() issubclass() iter()	<b>P</b> pow() print() property()	<b>Z</b> zip()
			<b>_</b> __import__()

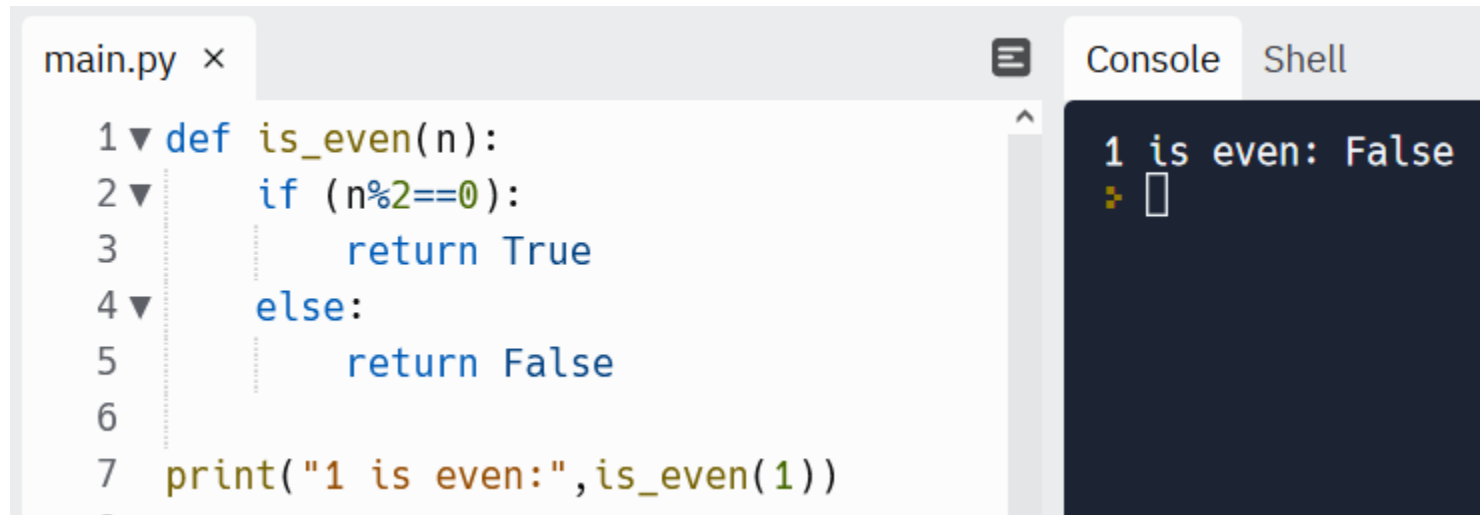
# Function

- รูปแบบของการเขียน Function



# Function

- กรณีที่มีการส่งค่ากลับ จะใช้คำสั่ง return

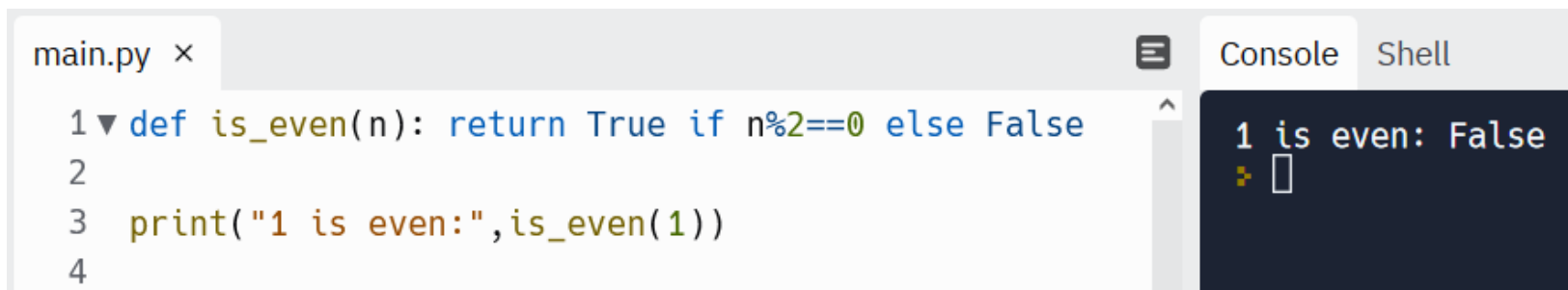


```
main.py x
1 ▼ def is_even(n):
2 ▼     if (n%2==0):
3         return True
4 ▼     else:
5         return False
6
7 print("1 is even:",is_even(1))
```

Console Shell

```
1 is even: False
>
```

- กรณีฟังก์ชันไม่ซับซ้อน จะเขียนย่อๆ ก็ได้



```
main.py x
1 ▼ def is_even(n): return True if n%2==0 else False
2
3 print("1 is even:",is_even(1))
4
```

Console Shell

```
1 is even: False
>
```

# Function

- สมมติเราจำเป็นต้องเขียน function `is_odd()` เพิ่มเติมจะเขียนอย่างไร
- เราจะเขียนแบบนี้ก็ได้

```
main.py ×  
1 ▼ def is_odd(n):  
2 ▼     if (n%2==1):  
3         return True  
4 ▼     else:  
5         return False
```

- แต่โปรแกรมมีลักษณะที่คล้ายกับ `is_even` ซึ่ง developer ที่ดีพึงหลีกเลี่ยง ดังนั้นเราจะเขียนแบบนี้ ซึ่งจะสั้นกว่า และมีการ reuse

```
7 ▼ def is_odd(n):  
8     return not(is_even(n))
```

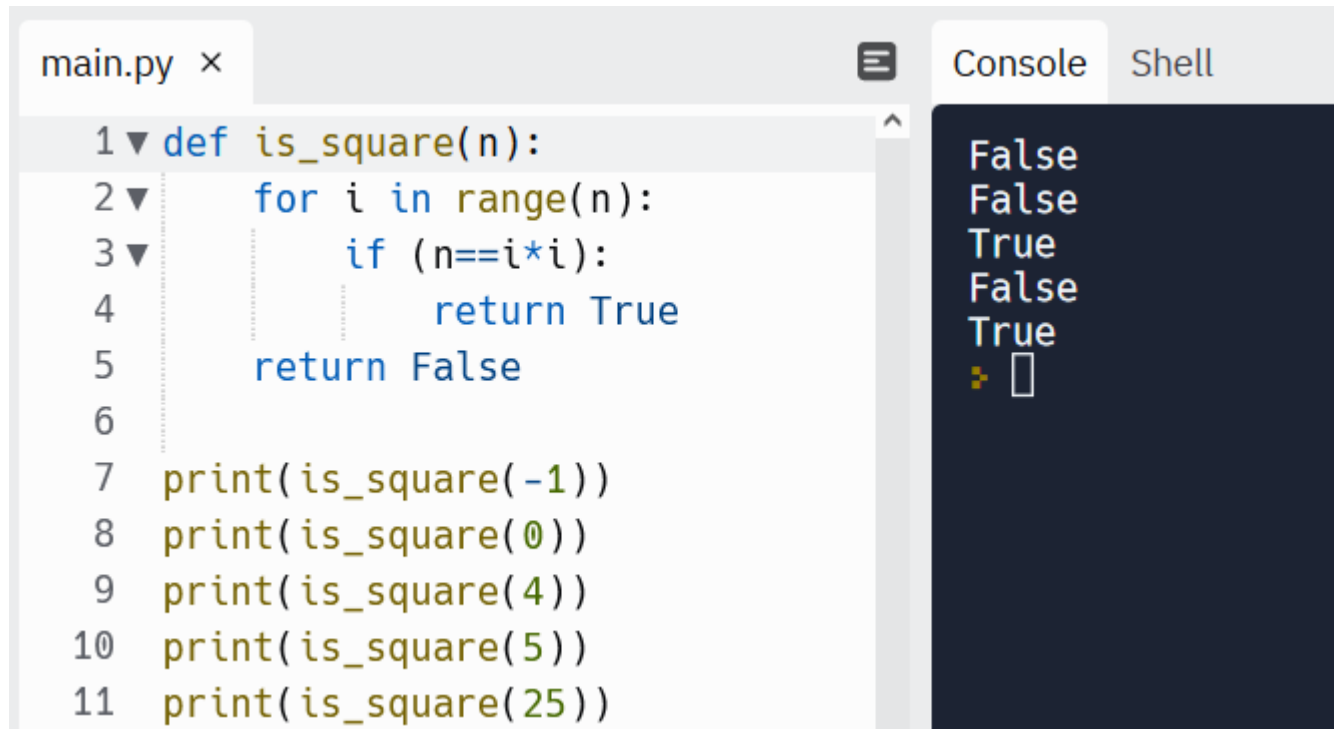
# Function

- การใช้ function จะทำให้โปรแกรมอ่านง่ายขึ้น (clean code) เช่น โปรแกรมที่แสดงเฉพาะเลขคู่ใน list ถ้าแยกส่วนตรวจสอบออกมา โปรแกรมจะดูง่ายขึ้น

```
main.py × +  
  
1 ▼ def is_even(n): return True if n%2==0 else False  
2  
3 ▼ def is_odd(n): return not is_even(n)  
4  
5 lst1 = [1,2,3,4,5,6,7,8,9]  
6  
7 ▼ for num in lst1:  
8 ▼     if is_even(num):  
9         print(num)  
10
```

# Function

- โปรแกรมรับค่าตัวเลข และบอกว่า เป็น square number หรือไม่ โดยทำเป็นฟังก์ชัน `is_square` คือ เป็นตัวเลขกำลังสองของเลขอื่นหรือไม่



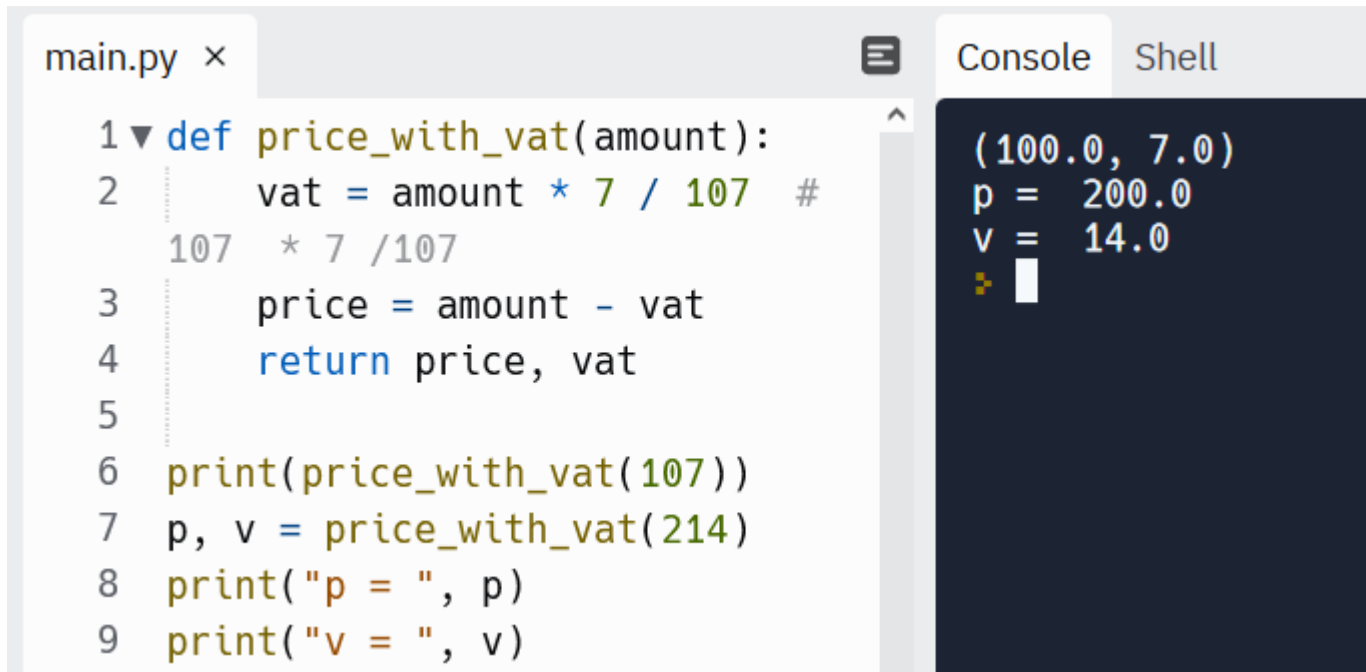
```
main.py x
1 def is_square(n):
2     for i in range(n):
3         if (n==i*i):
4             return True
5     return False
6
7 print(is_square(-1))
8 print(is_square(0))
9 print(is_square(4))
10 print(is_square(5))
11 print(is_square(25))
```

Console

```
False
False
True
False
True
>
```

# Function

- กรณีต้องการ Return หลายค่า
- เช่น จะเขียนโปรแกรมที่ใส่จำนวนเงินแล้ว Return มูลค่าสินค้า กับ ภาษีมูลค่าเพิ่ม



```
main.py x
1 ▼ def price_with_vat(amount):
2     vat = amount * 7 / 107 #
3     price = amount - vat
4     return price, vat
5
6 print(price_with_vat(107))
7 p, v = price_with_vat(214)
8 print("p = ", p)
9 print("v = ", v)
```

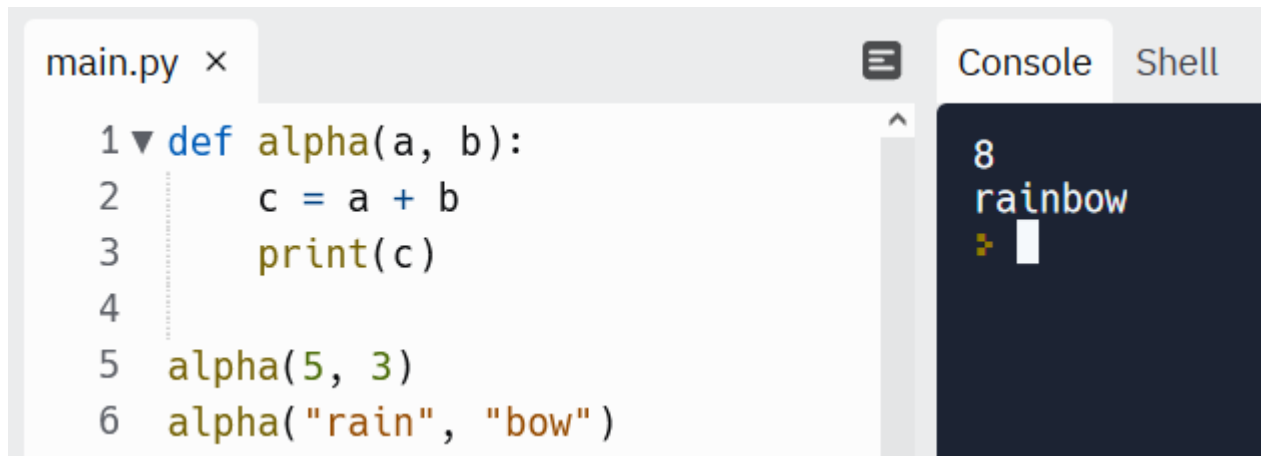
Console Shell

```
(100.0, 7.0)
p = 200.0
v = 14.0
```



# Function

- เนื่องจากตัวแปรของภาษา Python เป็น Duck Typing หรือ Dynamic Typing ดังนั้นต้องตรวจสอบโปรแกรมให้ดี ไม่เช่นนั้นอาจเกิดผลที่ไม่ต้องการได้



The screenshot shows a code editor with a file named 'main.py'. The code defines a function 'alpha(a, b)' that calculates the sum of 'a' and 'b' and prints the result. It then calls the function twice: first with numeric arguments (5, 3) and then with string arguments ('rain', 'bow'). The console output shows the results of these calls: '8' for the first call and 'rainbow' for the second call.

```
main.py x
1 def alpha(a, b):
2     c = a + b
3     print(c)
4
5 alpha(5, 3)
6 alpha("rain", "bow")
```

Console Shell

```
8
rainbow
>
```

- จะเห็นว่าฟังก์ชันทำงานถูกต้อง ในการเรียกใช้ทั้งสองครั้ง แต่ผลการทำงานต่างกัน โดยการเรียกครั้งแรกเป็นการบวก แต่การเรียกครั้งที่ 2 เป็นการ concatenate

# Function

- ในกรณีที่มี parameter หลายตัว มีความเป็นไปได้ว่าอาจใส่สลับกันมา เพื่อป้องกันไม่ให้ใส่สลับกัน อาจใช้วิธีระบุชื่อตัวแปรตอนเรียกก็ได้ (named argument)
- เราสามารถเริ่มใช้ named argument ตอนไหนก็ได้ แต่ argument หลังจากนั้นต้องเป็น named argument ด้วยทั้งหมด (ก่อนหน้านี้ ถือว่าเรียงตามลำดับ)

main.py ×

```
1 ▼ def price_with_vat2(amount, vat_rate):  
2     vat = amount * vat_rate / (100+vat_rate) # 107 * 7 /107  
3     price = amount - vat  
4     return price, vat  
5  
6 print(price_with_vat2(amount=107, vat_rate=7))  
7
```

Console

Shell

(100.0, 7.0)

# Function

- ในบางกรณีที่พารามิเตอร์บางตัวมักเป็นค่าใดค่าหนึ่งบ่อยๆ อาจกำหนดให้มีค่า default argument ได้
- จากรูปจะเห็นว่า vat\_rate มักจะเท่ากับ 7% ดังนั้นจึงกำหนดว่าถ้าไม่ได้ส่งค่าเข้าไป จะถือว่า = 7%
- แต่การใช้ default argument จะต้องเป็นพารามิเตอร์ตัวหลังสุดเท่านั้น

main.py ×

```
1 ▼ def price_with_vat2(amount, vat_rate=7):  
2     vat = amount * vat_rate / (100+vat_rate) # 107 * 7 /107  
3     price = amount - vat  
4     return price, vat  
5  
6 print(price_with_vat2(107))
```

Console Shell

```
(100.0, 7.0)  
✚ □
```

# Function

- บางฟังก์ชัน อาจไม่สามารถระบุจำนวน argument ที่แน่นอนได้ กรณีนี้จะเรียกว่า Arbitrary Arguments
- ภาษา Python มี feature ที่รองรับกรณีนี้ไว้ ตามแสดงในตัวอย่าง



The screenshot shows a code editor with a file named `main.py`. The code defines a function `greet(*names)` that takes an arbitrary number of arguments. The function's docstring states: `"""This function greets all the person in the names tuple."""`. Inside the function, a comment indicates that `names` is a tuple, and a `for` loop iterates over each name, printing `"Hello", name`. The function is then called with the arguments `"Monica", "Luke", "Steve", "John"`. To the right of the code editor, the 'Console' tab displays the output of the function: `Hello Monica`, `Hello Luke`, `Hello Steve`, and `Hello John`, followed by a prompt character `>`.

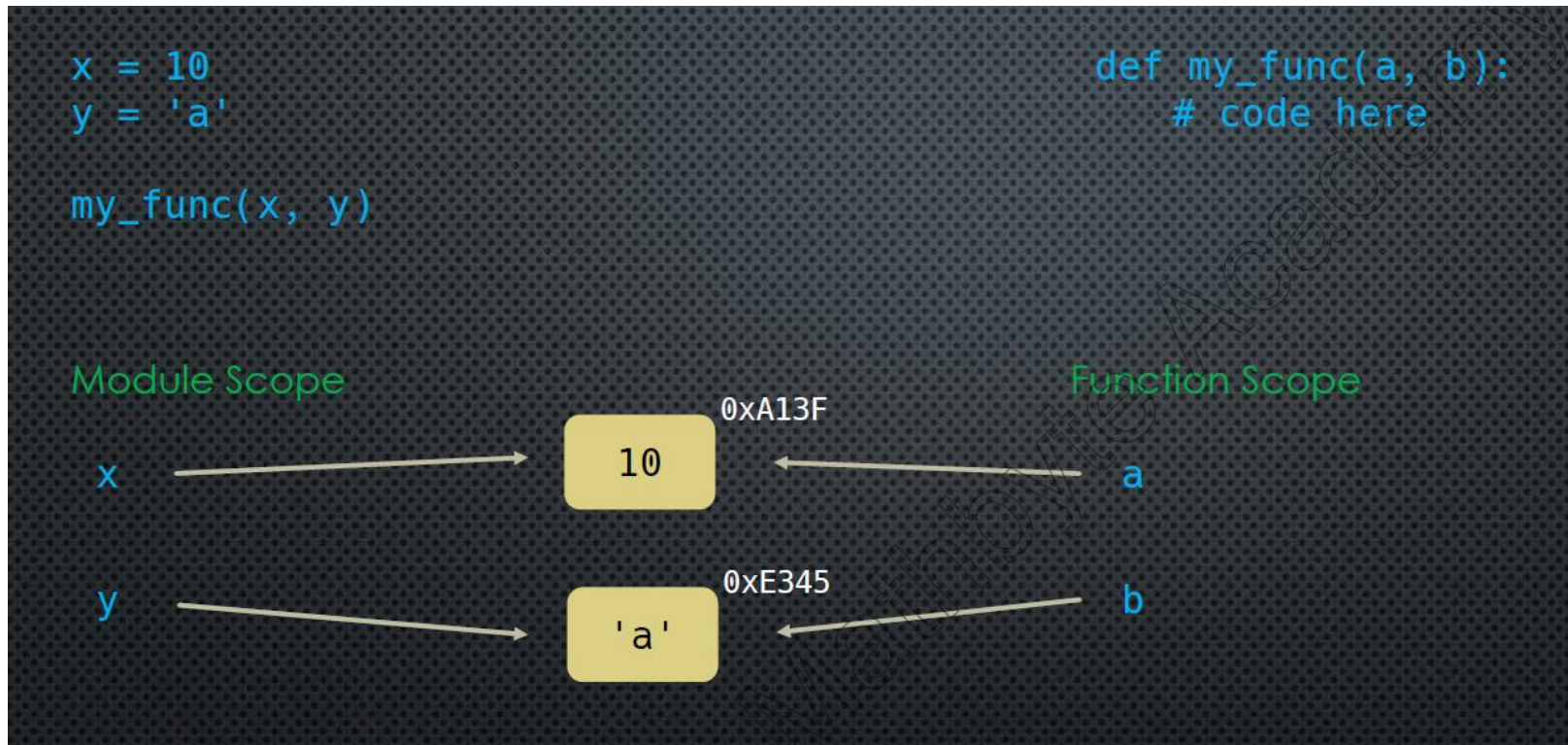
```
main.py ×  
1 ▼ def greet(*names):  
2     """This function greets all  
3     the person in the names tuple."""  
4  
5     # names is a tuple with arguments  
6 ▼   for name in names:  
7       print("Hello", name)  
8  
9  
10  greet("Monica", "Luke", "Steve", "John")  
11
```

Console  
Shell

```
Hello Monica  
Hello Luke  
Hello Steve  
Hello John  
>
```

# Function

- ในการส่งพารามิเตอร์เข้าไปในฟังก์ชัน เป็นการส่งตำแหน่งเข้าไปใน function เช่นจากรูปจะเห็นว่า argument ของฟังก์ชัน คือ a กับ b จะถูกกำหนดให้ชี้ไปที่ x และ y



# Function

- ตัวอย่าง เขียน function ชื่อ `day_of_year(day, month ,year)`  
โดยมีการคืนค่า คือ `day_of_years` เป็นวันที่ลำดับที่เท่าใดของปีคริสต์ศักราช `year`
  - ปีที่เป็น Leap Year เดือนกุมภาพันธ์จะมี 29 วัน
  - ให้สร้างฟังก์ชัน `is_leap` เพื่อตรวจสอบ leap year แยกออกมา และให้ฟังก์ชัน `day_of_year` เรียกใช้ `is_leap` อีกที

# Function

```
main.py × +
main.py
1  day_in_month = [0,31,28,31,30,31,30,31,31,30,31,30,31]
2
3 ▼ def is_leap(year):
4     return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
5
6 ▼ def day_of_year(day, month, year):
7     day_of_years = 0
8     if is_leap(year) :
9         day_in_month[2] += 1
10 ▼ else:
11 ▼     if month == 2 and day == 29:
12         return -1
13 ▼ for i in range(1,month):
14     print(day_in_month[i])
15     day_of_years += day_in_month[i]
16 day_of_years += day
17
18     return day_of_years
```

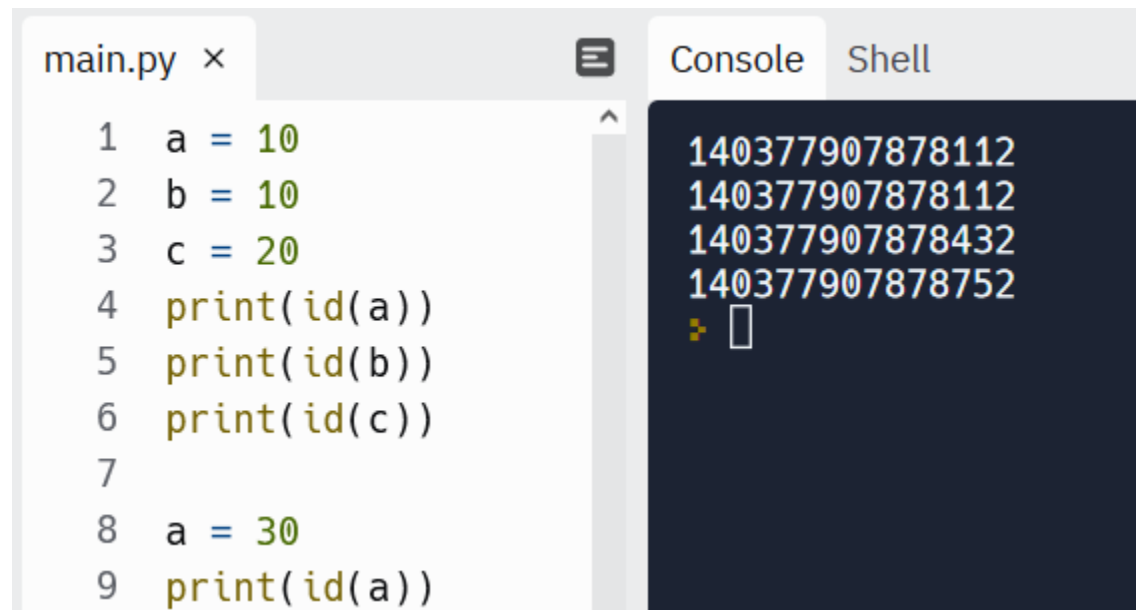
# Function

- การออกแบบโปรแกรมที่ดี พยายามให้โปรแกรมแต่ละส่วนสั้นและอ่านง่ายที่สุด ดังนั้นจึงต้องพยายามแยกส่วนโปรแกรมออกเป็น ฟังก์ชันย่อยๆ ให้มากที่สุด
- แต่ละฟังก์ชันควรมีหน้าที่เฉพาะ และ เสร็จในตัวเอง เช่น ฟังก์ชัน `is_leap` ที่แม้จะเป็นฟังก์ชันเล็กๆ แต่ก็ช่วยให้โปรแกรมอ่านง่ายขึ้น และ ซับซ้อนน้อยลง
- มีผู้ให้แนวทางปฏิบัติว่า แต่ละส่วนของโปรแกรมควรยาวประมาณ 15 บรรทัด จะทำให้โปรแกรมอ่านง่าย



# Mutable กับ Immutable

- List เป็นโครงสร้างข้อมูลที่เรียกว่า Mutable (แปลว่า เปลี่ยนแปลงได้)
- ส่วน Int, Float, Bool, String เป็นข้อมูลที่เป็น Immutable โดยหากเราแก้ไขข้อมูลในตัวแปรนั้น Python จะทำลายข้อมูลนั้น และสร้างขึ้นใหม่ โดยไม่ใช่ข้อมูลตำแหน่งเดิมตามตัวอย่างที่เมื่อเปลี่ยนค่าใน a จะพบว่า a จะไม่ได้อยู่ที่ตำแหน่งเดิม



The screenshot shows a Python IDE with a file named `main.py` and a console window. The code in `main.py` is as follows:

```
1 a = 10
2 b = 10
3 c = 20
4 print(id(a))
5 print(id(b))
6 print(id(c))
7
8 a = 30
9 print(id(a))
```

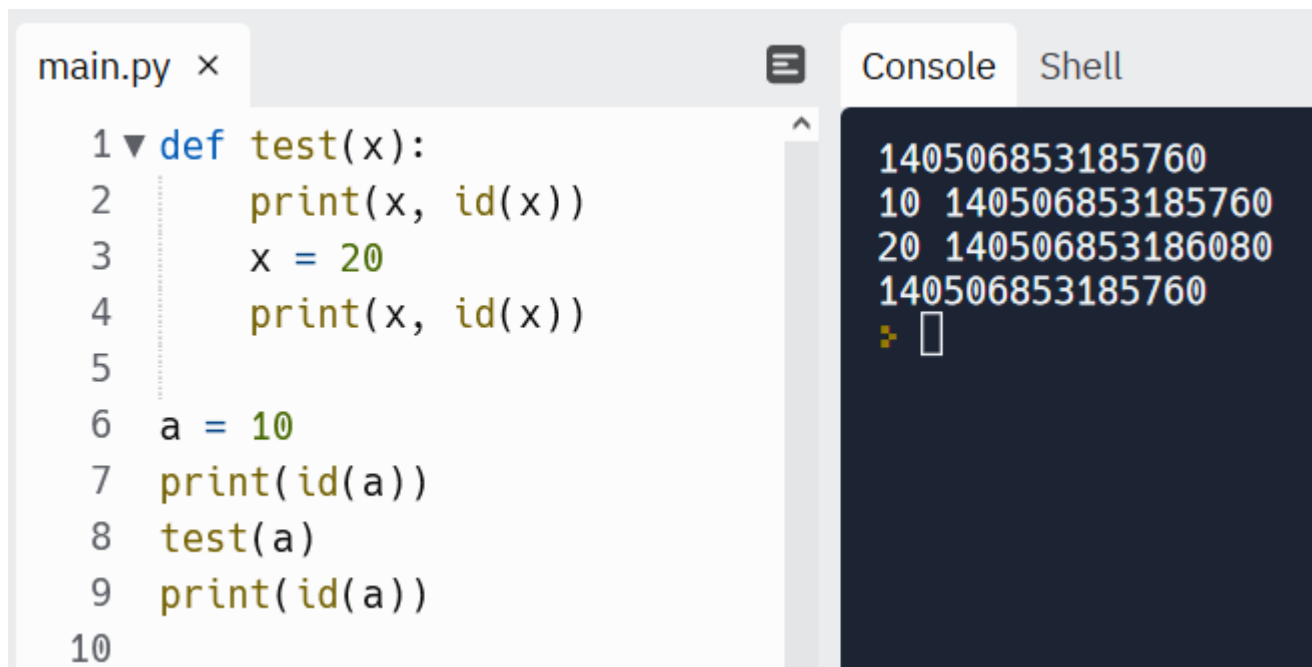
The console output shows the memory addresses (IDs) for each variable:

```
140377907878112
140377907878112
140377907878432
140377907878752
>
```

The output demonstrates that `a` and `b` share the same memory address (140377907878112) when both are 10. When `a` is changed to 30, it gets a new memory address (140377907878752), while `b` remains at the original address (140377907878112). This illustrates that integers are immutable in Python.

# Mutable กับ Immutable

- จากความรู้ข้างต้น ในกรณีที่เราส่งพารามิเตอร์เข้าไปใน argument ของฟังก์ชัน
- หากพารามิเตอร์นั้นเป็นแบบ Immutable ก็ไม่มีปัญหาอะไร เพราะหากมีการกำหนดค่าใหม่ในฟังก์ชัน ก็จะเหมือนกับเป็นตัวแปรใหม่ ไม่กระทบตัวแปรเดิม



The screenshot shows a Python IDE with a file named `main.py`. The code in the editor is as follows:

```
1 def test(x):  
2     print(x, id(x))  
3     x = 20  
4     print(x, id(x))  
5  
6 a = 10  
7 print(id(a))  
8 test(a)  
9 print(id(a))  
10
```

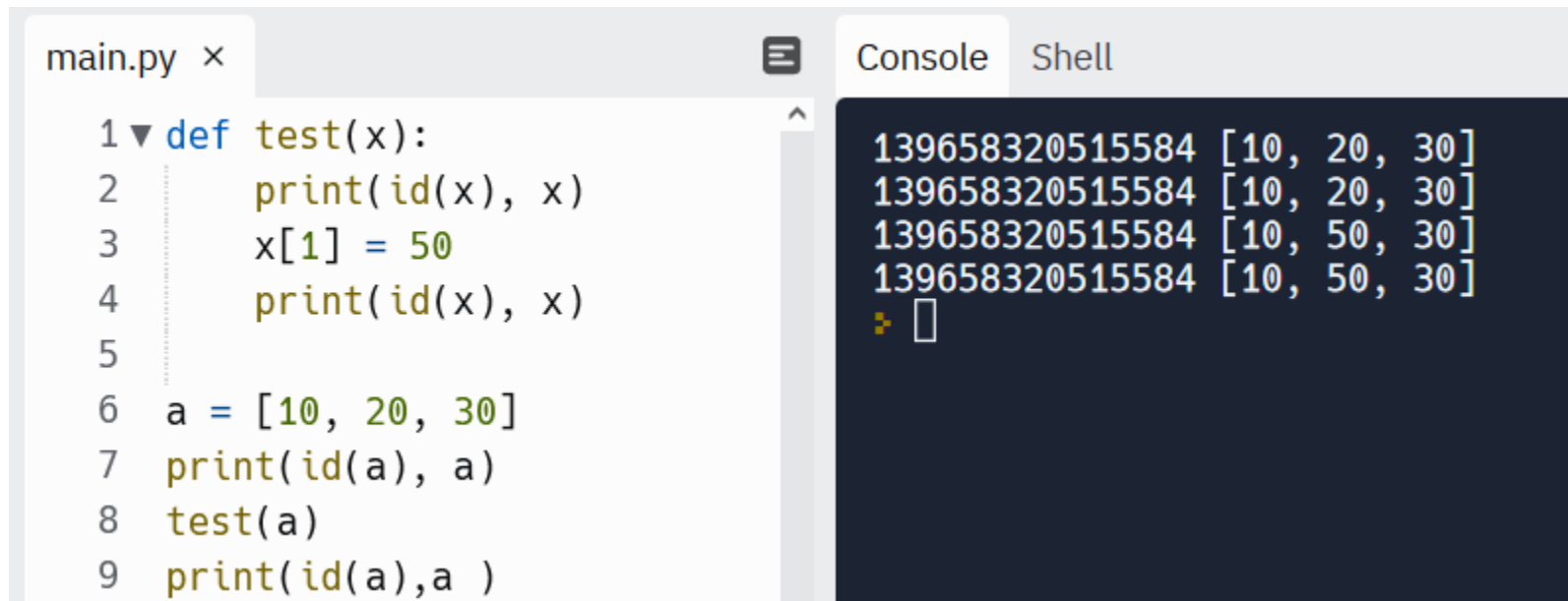
The console output on the right shows the execution results:

```
140506853185760  
10 140506853185760  
20 140506853186080  
140506853185760  
➤
```

The output demonstrates that for the immutable integer `10`, the memory address remains the same both before and after the function call. For the mutable integer `20`, the memory address changes after the function call, indicating that a new object was created.

# Mutable กับ Immutable

- แต่หากเป็นข้อมูลที่เป็น mutable แล้ว หากมีการแก้ไขข้อมูลตัวแปรภายในฟังก์ชัน อาจทำให้มีปัญหาดได้ (ยกเว้น กรณีที่เป็นความตั้งใจ)
- จะเห็นว่า List a ที่ส่งเป็นพารามิเตอร์ มีการแก้ไขไปด้วย ดังนั้นควรระวังกรณีนี้



The screenshot shows a Python IDE with a file named `main.py` and a console window. The script in `main.py` defines a function `test(x)` that prints the ID and value of `x`, modifies the second element of the list `x` to 50, and prints the ID and value again. It then creates a list `a = [10, 20, 30]`, prints its ID and value, calls `test(a)`, and prints the ID and value of `a` again.

```
1 def test(x):
2     print(id(x), x)
3     x[1] = 50
4     print(id(x), x)
5
6 a = [10, 20, 30]
7 print(id(a), a)
8 test(a)
9 print(id(a), a )
```

The console output shows the following:

```
139658320515584 [10, 20, 30]
139658320515584 [10, 20, 30]
139658320515584 [10, 50, 30]
139658320515584 [10, 50, 30]
>
```



*For your attention*