บทที่ 7

ฟังก์ชัน

เมื่อเขียนโปรแกรมไปสักระยะหนึ่งผู้อ่านพบว่าโปรแกรมมีขนาดใหญ่ขึ้น และรู้สึกว่ามีคำสั่งโปรแกรมบางส่วนทำงานเหมือนกัน ซึ่งเป็นสาเหตุทำให้เสียเวลาในการเขียนคำสั่งโปรแกรม นอกจากนี้ยังทำให้ต้องเสียเวลาเมื่อต้องการแก้ไขคำสั่งโปรแกรมเพราะต้องแก้ไขทุก ๆ จุด แต่มีวิธีการแก้ไขคือ นำคำสั่งโปรแกรมที่เขียนซ้ำกันบ่อย ๆ แยกออกมาสร้างเป็นโปรแกรมย่อย หรือที่เรียกว่า ฟังก์ชัน (Functions) ทำให้คำสั่งโปรแกรมของเรามีความเป็นระเบียบและแก้ไขได้ง่าย โดยฟังก์ชันแบ่งออกเป็น 2 ชนิด ได้แก่ ฟังก์ชันที่ผู้เขียนโปรแกรมสร้างขึ้นมาใช้งานเอง (User Defined Functions) และฟังก์ชันที่สร้างขึ้นมาจากผู้เขียนโปรแกรมท่านอื่นถูกเก็บไว้เป็นไลบรารี (Library) มีทั้งนำมาใช้ได้ฟรี (Open Source) และมีค่าใช้จ่าย (Commercial)

จากหลาย ๆ บทที่ผ่านมาผู้อ่านได้เรียกใช้งานฟังก์ชัน ซึ่งเป็นฟังก์ชันภายในภาษาไพธอน (Built-in Functions) ที่ภาษาไพธอนได้จัดเตรียมไว้ให้ใช้งาน เมื่อสั่งให้โปรแกรมทำงานภาษาไพธอนจะโหลดฟังก์ชันเหล่านี้เข้าสู่หน่วยความจำ เราสามารถตรวจสอบชื่อฟังก์ชันภายในได้โดยการใช้คำสั่ง `dir('__builtin__')` และเรียกใช้งานโดยไม่จำเป็นต้องใช้คำสั่ง `import`

ตัวอย่าง 7.1
การตรวจสอบชื่อฟังก์ชันภายในไพธอน (Built-in Functions)

```
print(dir('__builtins__'))
```

```
['__add__', '__class__', '__contains__', '__delattr__',
↪  '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
↪  '__getattribute__', '__getitem__', '__getnewargs__',
↪  '__gt__', '__hash__', '__init__', '__init_subclass__',
↪  '__iter__', '__le__', '__len__', '__lt__', '__mod__',
↪  '__mul__', '__ne__', '__new__', '__reduce__',
↪  '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
↪  '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
↪  'capitalize', 'casefold', 'center', 'count', 'encode',
↪  'endswith', 'expandtabs', 'find', 'format', 'format_map',
↪  'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
↪  'isdigit', 'isidentifier', 'islower', 'isnumeric',
↪  'isprintable', 'isspace', 'istitle', 'isupper', 'join',
↪  'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
↪  'removeprefix', 'removesuffix', 'replace', 'rfind',
↪  'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
↪  'split', 'splitlines', 'startswith', 'strip', 'swapcase',
↪  'title', 'translate', 'upper', 'zfill']
```

7.1 การสร้างฟังก์ชันขึ้นมาใช้งาน

การสร้างฟังก์ชันขึ้นมาใช้งาน (Creating Function) เป็นการกำหนดหน้าที่ให้กับคำสั่งโปรแกรมทำงานเฉพาะอย่าง เพื่อให้ได้ผลลัพธ์ตามที่ผู้เขียนโปรแกรมต้องการ มีรูปแบบการเขียนคำสั่งโปรแกรมสร้างฟังก์ชันดังต่อไปนี้

```
def function_name ([parameters]) :
    statements
    return [var, expression]
```

`def`	คำสั่งที่ใช้สร้างฟังก์ชัน
`function_name`	ชื่อฟังก์ชัน ห้ามตั้งชื่อซ้ำกับคำสงวน
`parameters`	ตัวแปรที่ใช้สำหรับรับค่าข้อมูล กำหนดหรือไม่ก็ได้ ถ้ากำหนดต้องมีเท่ากับค่าตัวแปรอาร์กิวเมนต์ที่ส่งมา
`statements`	ชุดคำสั่งการทำงานภายในฟังก์ชัน
`return`	คำสั่งคืนค่าผลลัพธ์กลับ ถ้าไม่มีการคืนค่ากลับไปยังโปรแกรมที่เรียกใช้งาน จะแสดงผลลัพธ์ด้วยคำว่า `None`
`var, expression`	ตัวแปรหรือนิพจน์ที่เก็บผลลัพธ์คืนค่ากลับ

จากรูปแบบการเขียนคำสั่งโปรแกรมสร้างฟังก์ชัน สามารถแบ่งวิธีการสร้างและวิธีการทำงานของฟังก์ชันออกเป็น 4 รูปแบบดังต่อไปนี้

7.1.1 การสร้างฟังก์ชันที่ไม่มีการส่งค่าและรับค่า

เป็นการสร้างฟังก์ชันที่ไม่มีพารามิเตอร์คอยรับอาร์กิวเมนต์ หลังจากฟังก์ชันทำงานเสร็จก็จะไม่มีการส่งค่ากลับคืนไปยังโปรแกรมที่เรียกใช้งาน มีรูปแบบการเขียนคำสั่งโปรแกรมดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.2
การเขียนคำสั่งโปรแกรมแบบไม่มีการส่งค่าและรับค่า

```
1 def show_greeting1(): # ประกาศฟังก์ชันโดยไม่มีพารามิเตอร์คอยรับค่า
2     print('Hello') # แสดงผลลัพธ์
3
4 show_greeting1() # เรียกใช้งานฟังก์ชัน show_greeting1()
```

```
Hello
```

จากตัวอย่างเมื่อสั่งให้โปรแกรมทำงาน บรรทัดที่ 3 จะเรียกฟังก์ชันให้แสดงผลคำว่า `'Hello'`

7.1.2 การสร้างฟังก์ชันที่มีการส่งค่าและรับค่า แต่ไม่มีการส่งค่ากลับ

เป็นการสร้างฟังก์ชันที่มีพารามิเตอร์คอยรับอาร์กิวเมนต์ หลังจากฟังก์ชันทำงานเสร็จจะไม่มีการส่งค่ากลับไปยังโปรแกรมที่เรียกใช้งาน มีรูปแบบการเขียนคำสั่งโปรแกรมดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.3
การเขียนคำสั่งโปรแกรมแบบสร้างฟังก์ชัน โดยมีการส่งค่าและรับค่า แต่ไม่มีการส่งค่ากลับ

```
1 def show_greeting2(name): # ประกาศฟังก์ชันโดยมีพารามิเตอร์คอยรับค่า
2     print('Hello', name) # แสดงผลลัพธ์จากค่าพารามิเตอร์
3
4 show_greeting2('Doraemon') # เรียกใช้งานฟังก์ชันพร้อมส่งอาร์กิวเมนต์
```

```
Hello Doraemon
```

ฟังก์ชัน `show_greeting2()` แสดงผลการทักทายผู้ใช้งานคล้ายกับตัวอย่างที่ 10.2 แต่เมื่อเรียกใช้งานฟังก์ชันในบรรทัดที่ 3 มีการส่งค่าอาร์กิวเมนต์คือคำว่า `'Doraemon'` ไปให้

กับฟังก์ชันที่มีค่าพารามิเตอร์ชื่อ `name` คอยรับค่า จากนั้นฟังก์ชันจะนำพารามิเตอร์แสดงผลร่วมกับคำว่า `'Hello'` ในบรรทัดที่ 2

7.1.3 การสร้างฟังก์ชันที่ไม่มีการส่งค่าและรับค่า แต่มีการส่งค่ากลับ

มีวิธีการสร้างฟังก์ชันขึ้นมาใช้งานเหมือนกับวิธีแรก แต่เมื่อฟังก์ชันทำงานเสร็จจะมีการส่งค่ากลับด้วยคำสั่ง **`return`** มีรูปแบบการเขียนคำสั่งโปรแกรมดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.4
การเขียนคำสั่งโปรแกรมแบบไม่มีการส่งค่าและรับค่า แต่ส่งค่ากลับด้วยคำสั่ง `return`

```
1 def show_greeting3(): # ประกาศฟังก์ชันโดยไม่มีพารามิเตอร์คอยรับค่า
2     greeting = 'Hello, how are you?' # สร้างตัวแปรพร้อมกำหนดค่า
3     return greeting # ส่งค่าตัวแปร greeting กลับ เมื่อมีการเรียกใช้งานฟังก์ชัน
4
5 print(show_greeting3()) # แสดงผลลัพธ์จากการเรียกใช้งานฟังก์ชัน
```

```
Hello, how are you?
```

จากตัวอย่างโปรแกรมเป็นการเรียกใช้งานฟังก์ชัน `show_greeting3()` แต่ไม่มีการส่งค่าอาร์กิวเมนต์ให้กับฟังก์ชัน และฟังก์ชันก็ไม่มีพารามิเตอร์คอยรับค่า แต่ฟังก์ชันสามารถส่งค่ากลับมายังโปรแกรมที่เรียกใช้งานได้ โดยการใช้คำสั่ง **`return`** จากตัวอย่างจะคืนค่าตัวแปร `greeting` กลับมาแสดงผล

7.1.4 การสร้างฟังก์ชันที่มีการส่งค่าและรับค่า พร้อมทั้งมีการส่งค่ากลับ

เป็นการสร้างฟังก์ชันที่มีพารามิเตอร์คอยรับอาร์กิวเมนต์ หลังจากฟังก์ชันทำงานเสร็จก็จะคืนค่ากลับมายังโปรแกรมที่เรียกใช้งานด้วยคำสั่ง **`return`** มีรูปแบบการเขียนคำสั่งโปรแกรมดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.5
การเขียนคำสั่งโปรแกรมที่มีการส่งค่า-รับค่า และส่งค่ากลับด้วยคำสั่ง `return`

```
1 def show_greeting4(name): # ประกาศฟังก์ชันโดยมีพารามิเตอร์คอยรับค่า
2     # สร้างตัวแปรพร้อมกำหนดค่า
3     greeting_name ='Hello, ' + name + '. ' + 'Where are you?'
4     # ส่งค่าตัวแปร greeting กลับ เมื่อมีการเรียกใช้งานฟังก์ชัน
5     return greeting_name
6
7 # แสดงผลลัพธ์จากการเรียกใช้งานฟังก์ชันพร้อมมีการส่งค่า
8 print(show_greeting4('Doraemon'))
```

```
Hello, Doraemon. Where are you?
```

จากตัวอย่างโปรแกรมมีการส่งอาร์กิวเมนต์ไปให้กับพารามิเตอร์ของฟังก์ชัน `show_greeting4()` โดยฟังก์ชันจะเก็บข้อมูลรวมกันไว้ที่ตัวแปร `greeting_name` จากนั้นใช้คำสั่ง **`return`** ส่งกลับมายังคำสั่งโปรแกรมที่เรียกใช้งานฟังก์ชัน

7.2 อาร์กิวเมนต์และพารามิเตอร์

จากหลาย ๆ ตัวอย่างการเขียนโปรแกรมแบบฟังก์ชัน ผู้อ่านจะได้พบคำศัพท์อยู่ 2 คำคือ อาร์กิวเมนต์ และพารามิเตอร์ บางท่านอาจจะยังสงสัยอยู่ว่าทั้งสองทำหน้าที่อะไรบ้าง

- อาร์กิวเมนต์ (Argument) คือ ค่าตัวแปรหรือค่าคงที่ซึ่งอ้างถึงในตอนเรียกใช้สำหรับส่งค่าไปให้กับฟังก์ชันที่มีการเรียกใช้งาน ค่าอาร์กิวเมนต์จะถูกแนบสงไปพร้อมกับชื่อฟังก์ชันอยู่ในเครื่องหมายวงเล็บ หากมีอาร์กิวเมนต์หลายตัวจะคั่นด้วยเครื่องหมาย Comma **`(,)`**
- พารามิเตอร์ (Parameter) คือ ตัวแปรที่ประกาศไว้ในฟังก์ชันเพื่อรอรับค่าอาร์กิวเมนต์อยู่ในเครื่องหมายวงเล็บหลังชื่อฟังก์ชัน และคั่นด้วยเครื่องหมาย Comma **`(,)`** ถ้ามีพารามิเตอร์มากกว่าหนึ่งตัว

ภาพแสดงความสัมพันธ์ระหว่างอาร์กิวเมนต์กับพารามิเตอร์

เมื่อมีการเรียกใช้งานฟังก์ชันและส่งค่าอาร์กิวเมนต์ไปให้พารามิเตอร์ที่ประกาศไว้รอรับค่า จำนวนอาร์กิวเมนต์และจำนวนพารามิเตอร์จะต้องเท่ากัน ในการส่งค่าอาร์กิวเมนต์มีอยู่ด้วย 2 วิธีด้วยกันคือ การส่งค่าข้อมูล (Call by Value หรือ Pass by Value) และการส่งค่าแบบอ้างอิง (Call by Reference หรือ Pass by Reference)

- Call by Value หรือ Pass by Value ค่าอาร์กิวเมนต์จะถูกทำการ copy แล้วส่งให้กับพารามิเตอร์ของฟังก์ชันที่เราเรียกใช้งาน จำนวนพารามิเตอร์ต้องเท่ากับจำนวนอาร์กิวเมนต์ที่ส่งมาให้ การเปลี่ยนแปลงค่าพารามิเตอร์ภายในฟังก์ชัน จะไม่มีผลกระทบต่อค่าอาร์กิวเมนต์เมื่อมีการส่งค่ากลับ การส่งค่าอาร์กิวเมนต์แบบนี้ทำให้สิ้นเปลืองเนื้อที่ในหน่วยความจำ

- Call by Reference หรือ Pass by Reference ใช้กับข้อมูลชนิดออบเจ็ค โดยส่งค่าตำแหน่งอาร์กิวเมนต์ในหน่วยความจำ ไปให้กับพารามิเตอร์ในฟังก์ชันที่เรียกใช้งาน เมื่อมีการเปลี่ยนแปลงข้อมูลของค่าพารามิเตอร์จะทำให้ค่าอาร์กิวเมนต์ที่ส่งไปเปลี่ยนตามไปด้วย การส่งอาร์กิวเมนต์แบบนี้มีข้อดีคือ ประหยัดเนื้อที่ในหน่วยความจำ เพราะค่าอาร์กิวเมนต์ไม่ต้องมีการ copy อีกชุดหนึ่งเหมือนกับวิธีแรก และลดเวลาในการประมวลผลข้อมูล

นอกจากนี้ในภาษาไพธอนยังมีกลไกการส่งอาร์กิวเมนต์ที่เรียกว่า Call by Object บางครั้งเรียกว่า Call by Object Reference หรือ Call by Sharing ถ้าหากเราส่งค่าอาร์กิวเมนต์เป็นชนิดข้อมูลที่ไม่สามารถเปลี่ยนรูปได้ (Immutable) เช่น ชนิดข้อมูลจำนวนเต็ม สตริง ทูเพิล เป็นต้น ไปยังฟังก์ชันผ่านค่าพารามิเตอร์ หากมีการเปลี่ยนค่าพารามิเตอร์ภายในฟังก์ชัน จะไม่มีผลกระทบต่อค่าอาร์กิวเมนต์ที่ต้นทาง คือ มีลักษณะการทำงานเหมือนกับ Call by Value แต่หากเราส่งค่าอาร์กิวเมนต์ที่เป็นชนิดข้อมูลลิสต์ ซึ่งเป็นชนิดข้อมูลที่เปลี่ยนแปลงได้ (Mutable) จะส่งผลให้เมื่อมีการเปลี่ยนแปลงค่าพารามิเตอร์ภายในฟังก์ชัน ค่าอาร์กิวเมนต์ต้นทางก็จะถูกเปลี่ยนค่าตามไปด้วย ดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 7.6
การเขียนคำสั่งโปรแกรมที่มีการส่งค่า-รับค่าแบบ Call by Value

```
1  def msg(f_str): # ประกาศฟังก์ชันโดยมีพารามิเตอร์คอยรับค่า
2      # แสดงผลลัพธ์ค่าพารามิเตอร์
3      print('ค่าพารามิเตอร์ที่รับมา =', f_str)
4      # เปลี่ยนค่าข้อมูลพารามิเตอร์ f_str
5      f_str = 'EASY'
6      # แสดงผลลัพธ์หลังจากแปลี่ยนค่าพารามิเตอร์
7      print('เปลี่ยนค่าพารามิเตอร์ = ', f_str)
8
9
10 # กำหนดค่าให้กับตัวแปร
11 arg_str ='PYTHON'
12 # แสดงผลลัพธ์ค่าตัวแปร arg_str ก่อนส่งค่าให้กับฟังก์ชัน
13 print('ค่าอาร์กิวเมนต์ที่ส่งให้ค่าพารามิเตอร์ =', arg_str)
14 # ส่งค่าอาร์กิวเมนต์ให้กับฟังก์ชัน
15 msg(arg_str)
16 # แสดงผลลัพธ์ค่าตัวแปร arg_str หลังจากฟังก์ชันส่งกลับ
17 print ('ค่าอาร์กิวเมนต์หลังจากเปลี่ยนค่าพารามิเตอร์ส่งกลับ =', arg_str)
```

```
ค่าอาร์กิวเมนต์ที่ส่งให้ค่าพารามิเตอร์ = PYTHON
ค่าพารามิเตอร์ที่รับมา = PYTHON
เปลี่ยนค่าพารามิเตอร์ =  EASY
ค่าอาร์กิวเมนต์หลังจากเปลี่ยนค่าพารามิเตอร์ส่งกลับ = PYTHON
```

จากตัวอย่างโปรแกรมภายในฟังก์ชันมีการเปลี่ยนค่าพารามิเตอร์ `f_str` ในบรรทัดที่ 3 จะมีผลทำให้เปลี่ยนแปลงเฉพาะภายในฟังก์ชันเท่านั้น ไม่กระทบถึงตัวแปร `arg_str`

ตัวอย่าง 7.7
การเขียนคำสั่งโปรแกรมส่งค่า-รับค่า และมีการเปลี่ยนแปลงข้อมูล

```
1  def msg(f_lst):
2      print('ค่าพารามิเตอร์ที่รับมา = ', f_lst)
3      f_lst.append('EASY')
4      print('เปลี่ยนค่าพารามิเตอร์ = ', f_lst)
5
6  arg_lst = ['PYTHON IS']
7  print('ค่าอาร์กิวเมนต์ที่ส่งให้ค่าพารามิเตอร์ =', arg_lst)
8  msg(arg_lst)
9  print('ค่าอาร์กิวเมนต์หลังจากเปลี่ยนค่าพารามิเตอร์ =', arg_lst)
```

```
ค่าอาร์กิวเมนต์ที่ส่งให้ค่าพารามิเตอร์ = ['PYTHON IS']
ค่าพารามิเตอร์ที่รับมา =  ['PYTHON IS']
เปลี่ยนค่าพารามิเตอร์ =  ['PYTHON IS', 'EASY']
ค่าอาร์กิวเมนต์หลังจากเปลี่ยนค่าพารามิเตอร์ = ['PYTHON IS', 'EASY']
```

จากตัวอย่างโปรแกรมผู้อ่านจะพบว่า เมื่อมีการส่งค่าอาร์กิวเมนต์ที่เป็นชนิดข้อมูลลิสต์ซึ่งเป็นชนิดข้อมูลที่สามารถแก้ไขข้อมูลได้ ส่งไปให้กับฟังก์ชันที่มีพารามิเตอร์คอยรับค่าอยู่ และภายในฟังก์ชันมีเปลี่ยนแปลงค่าพารามิเตอร์ จากตัวอย่างในบรรทัดที่ 3 ใช้เมธอด `append()` เพิ่มข้อมูลต่อท้ายลิสต์ หลังจากสั่งให้โปรแกรมแสดงผลในบรรทัดที่ 4 ค่าพารามิเตอร์จะแสดงผลค่าข้อมูลที่เพิ่มเข้าไปในฟังก์ชันด้วย และเมื่อแสดงผลค่าอาร์กิวเมนต์ในบรรทัดที่ 8 จะเห็นว่าค่าอาร์กิวเมนต์ก็จะเปลี่ยนตามค่าพารามิเตอร์ที่ถูกเปลี่ยนแปลง

7.3 รูปแบบการส่งค่าอาร์กิวเมนต์ให้กับค่าพารามิเตอร์

การสร้างฟังก์ชันขึ้นมาใช้งานเพื่อทำหน้าที่อย่างใดอย่างหนึ่งนั้น มีรูปแบบการส่งค่าอาร์กิวเมนต์และการกำหนดค่าพารามิเตอร์ของฟังก์ชันที่ถูกเรียกใช้งานได้หลากหลายวิธี ซึ่งผู้อ่านจะได้ศึกษาและนำไปประยุกต์ใช้งานในการพัฒนาโปรแกรมในหัวข้อนี้

7.3.1 การส่งค่าอาร์กิวเมนต์แบบ Required argumants

คือวิธีการส่งค่าอาร์กิวเมนต์ไปให้กับค่าพารามิเตอร์ในฟังก์ชันที่ถูกเรียกใช้งาน โดยจำนวนของค่าอาร์กิวเมนต์ต้องเท่ากับจำนวนพารามิเตอร์ที่มีอยู่ ซึ่งค่าพารามิเตอร์จะแสดงผลตามชนิดข้อมูลที่ค่าอาร์กิวเมนต์ส่งให้ หากจำนวนอาร์กิวเมนต์และพารามิเตอร์ไม่เท่ากันจะทำให้เกิดการแจ้งเตือนข้อผิดพลาด ผู้เขียนโปรแกรมอาจจะตั้งชื่ออาร์กิวเมนต์และพารามิเตอร์เหมือนหรือต่างกันก็ได้ แต่ขอแนะนำให้ตั้งชื่อเหมือนกันและให้สื่อความหมายจะดีกว่า ดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 7.8
การส่งค่าอาร์กิวเมนต์แบบ Required arguments ไปให้กับค่าพารามิเตอร์

```
def req_arg(numlst, str_, num):
    print('แสดงค่าข้อมูลในลิสต์ =', numlst)
    print('แสดงค่าข้อมูลในตัวแปร num =', num)
    print('แสดงค่าข้อมูลในตัวแปร str_ =', str_)

lst = [1,2,3]
msg = 'Python'
req_arg(lst, msg, 50)
```

```
แสดงค่าข้อมูลในลิสต์ = [1, 2, 3]
แสดงค่าข้อมูลในตัวแปร num = 50
แสดงค่าข้อมูลในตัวแปร str_ = Python
```

จากตัวอย่างเราสามารถส่งค่าอาร์กิวเมนต์ไปให้กับค่าพารามิเตอร์ได้หลายชนิดข้อมูล แต่ต้องมีจำนวนเท่ากันและต้องเรียงลำดับให้ถูกต้องด้วย

7.3.2 การส่งอาร์กิวเมนต์แบบ Keyword arguments

เป็นวิธีการส่งค่าอาร์กิวเมนต์ไปให้กับค่าพารามิเตอร์ที่ฟังก์ชันถูกเรียกใช้งาน ที่ไม่จำเป็นต้องเรียงลำดับชื่อของค่าพารามิเตอร์ตามชนิดข้อมูลที่ค่าอาร์กิวเมนต์ส่งให้ แต่มีข้อแม้ว่าชื่อของค่าอาร์กิวเมนต์กับค่าพารามิเตอร์ต้องมีชื่อเหมือนกัน และค่าทั้งสองต้องมีจำนวนเท่ากันด้วย

ตัวอย่าง 7.9
การส่งค่าอาร์กิวเมนต์แบบ Keyword arguments

```
1  def key_arg(lst, msg, num):
2      print('แสดงค่าข้อมูลในลิสต์ =', lst)
3      print('แสดงค่าข้อมูลในตัวแปร num =', num)
4      print('แสดงค่าข้อมูลในตัวแปร msg =', msg)
5
6  key_arg(lst=[1, 2, 3], num=10, msg='Python is easy')
```

```
แสดงค่าข้อมูลในลิสต์ = [1, 2, 3]
แสดงค่าข้อมูลในตัวแปร num = 10
แสดงค่าข้อมูลในตัวแปร msg = Python is easy
```

จากตัวอย่างจะเห็นว่าค่าอาร์กิวเมนต์และค่าพารามิเตอร์มีชื่อเหมือนกัน แต่มีการสลับตำแหน่งชื่อค่าพารามิเตอร์ในฟังก์ชัน เมื่อโปรแกรมทำงานและส่งค่าอาร์กิวเมนต์ไปให้กับค่าพารามิเตอร์ก็จะอ้างอิงตามชื่อ

7.3.3 การส่งอาร์กิวเมนต์แบบ Default arguments

มีค่าพารามิเตอร์บางตัวได้ถูกกำหนดค่าไว้ล่วงหน้า โดยเมื่อเรียกใช้ก็จะส่งค่าอาร์กิวเมนต์เฉพาะจำนวนที่เหลือไปให้กับค่าพารามิเตอร์กับฟังก์ชันที่ถูกเรียกใช้งานเท่านั้น

ตัวอย่าง 7.10
การหาค่าเส้นรอบวงกลมโดยมีการกำหนดค่าคงที่ให้กับค่าพารามิเตอร์ไว้

```
1 def area_circle(r, pi=3.14):
2     result = 2 * pi * r
3     print('ความยาวเส้นรอบวงกลม = ', result)
4
5 area_circle(7)
6 area_circle(7, 3.1415926)
7 area_circle(7, pi=3.1415926)
```

```
ความยาวเส้นรอบวงกลม =  43.96
ความยาวเส้นรอบวงกลม =  43.9822964
ความยาวเส้นรอบวงกลม =  43.9822964
```

จากตัวอย่างโปรแกรมได้กำหนดพารามิเตอร์ชื่อ `pi=3.14` และมีพารามิเตอร์ชื่อ `r` ซึ่งเป็นค่ารัศมีที่คอยรับค่าจากอาร์กิวเมนต์ เมื่อเรียกใช้งานฟังก์ชัน `def_arg()` ก็ส่งอาร์กิวเมนต์เพียงค่าเดียวไปให้พารามิเตอร์ `r` ผลลัพธ์ที่แสดงออกมาคือขนาดของเส้นรอบวงกลม

7.3.4 การส่งอาร์กิวเมนต์แบบ Variable-length arguments

เป็นการสร้างพารามิเตอร์ไว้คอยรับอาร์กิวเมนต์แบบไม่จำกัดจำนวน โดยชื่อของพารามิเตอร์ประเภทนี้จะมีเครื่องหมาย `(*)` นำหน้า เมื่อต้องการนำข้อมูลในพารามิเตอร์มาแสดงผลหรือนำไปประมวลผล ให้ระบุตำแหน่งด้วยเครื่องหมาย `[...]` เพราะเป็นพารามิเตอร์ชนิดข้อมูลทูเพิล

ตัวอย่าง 7.11
การกำหนดพารามิเตอร์แบบ Variable-length arguments

```
def varleng_arg(*num):
    print('จำนวนข้อมูลในตัวแปร num :')
    for var in num:
        print(var, end=' ')
    print(' ')
    result = num[2] * 2
    print(f'ผลคูณพารามิเตอร์ num[2] * 2 = {result}')

print('-' * 50)
varleng_arg(10, 20, 30, 40)
print('-' * 50)
varleng_arg(2, 3, 5, 7, 11, 13, 17, 19)
print('-' * 50)
```

```
--------------------------------------------------
จำนวนข้อมูลในตัวแปร num :
10 20 30 40
ผลคูณพารามิเตอร์ num[2] * 2 = 60
--------------------------------------------------
จำนวนข้อมูลในตัวแปร num :
2 3 5 7 11 13 17 19
ผลคูณพารามิเตอร์ num[2] * 2 = 10
--------------------------------------------------
```

เมื่อโปรแกรมทำงานจะเรียกใช้งานฟังก์ชัน `varleng_arg()` ที่มีพารามิเตอร์ `*num` รองรับอาร์กิวเมนต์ที่ส่งมาให้แบบไม่จำกัด จากตัวอย่างโปรแกรมมีการส่งอาร์กิวเมนต์ไปให้พารามิเตอร์ทั้งหมด 4 ค่า เราสามารถแสดงผลค่าข้อมูลในพารามิเตอร์ด้วยคำสั่ง `for` และนำข้อมูลในพารามิเตอร์มาประมวลด้วยการระบุตำแหน่งด้วยเครื่องหมาย `[...]`

7.4 การสร้างฟังก์ชันด้วยคำสั่ง lambda

โดยปกติการสร้างฟังก์ชันขึ้นมาใช้งานจะใช้คำสั่ง **def** และมีการตั้งชื่อให้แต่ละฟังก์ชันนั้น ๆ แต่เราสามารถที่จะสร้างฟังก์ชันขึ้นมาใช้งานโดยไม่ต้องกำหนดชื่อ ด้วยการเรียกใช้คำสั่ง **lambda** (แลมป์ดา) โดยมีรูปแบบการใช้งานดังนี้

```
var = lambda argument_lists ; expression
```

var	ตัวแปรที่คอยรับค่าจากการดำเนินของฟังก์ชัน **lambda**
argument_lists	อาร์กิวเมนต์ส่งค่าไปประมวลผล
expression	นิพจน์ที่ใช้สำหรับการประมวลผล

ตัวอย่าง 7.12
การสร้างฟังก์ชัน lambda หาปริมาตรทรงกระบอก

```
1 volume = lambda r, h: 3.14 * (r * r) * h
2 print(f'ปริมาตรทรงกระบอกที่ 1 = {volume(2, 3)}')
3 print(f'ปริมาตรทรงกระบอกที่ 2 = {volume(3, 5)}')
4 print(f'ปริมาตรทรงกระบอกที่ 3 = {volume(4, 12)}')
```

```
ปริมาตรทรงกระบอกที่ 1 = 37.68
ปริมาตรทรงกระบอกที่ 2 = 141.3
ปริมาตรทรงกระบอกที่ 3 = 602.88
```

จากตัวอย่างเป็นการสร้างฟังก์ชันด้วยคำสั่ง **lambda** เมื่อเปรียบเทียบกับวิธีการสร้างฟังก์ชันโดยใช้คำสั่ง **def** จะมีขนาดคำสั่งที่กระทำรัดกว่า และอาร์กิวเมนต์หลังคำสั่ง **lambda** จะมีกี่ตัวก็ได้ ซึ่งถูกคั่นด้วยเครื่องหมาย Comma **(,)** และหลังอาร์กิวเมนต์ตัวสุดท้ายจะถูกปิดท้ายด้วยเครื่องหมาย Colon **(:)** ก่อนเขียนนิพจน์การคำนวณค่า

การสร้างฟังก์ชันด้วยคำสั่ง **lambda** ไม่จำเป็นต้องส่งค่าอาร์กิวเมนต์ก็ได้ โดยจะมีเฉพาะนิพจน์สำหรับประมวลผลข้อมูลเท่านั้น ดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.13
การสร้างฟังก์ชันด้วยคำสั่ง lambda ที่ไม่มีการส่งค่าอาร์กิวเมนต์

```
1 volume = lambda: 3.14 * 5 * 5 * 10
2 print(f'ปริมาตรทรงกระบอก = {volume()}')
```

```
ปริมาตรทรงกระบอก = 785.0
```

จากตัวอย่างโปรแกรมเป็นการสร้างปริมาตรทรงกระบอกด้วยฟังก์ชัน **lambda** แต่ไม่มีการส่งค่าอาร์กิวเมนต์ให้กับฟังก์ชัน **lambda** เราแสดงผลลัพธ์จากการคำนวณของฟังก์ชันในค่าตัวแปร `volume()` ได้เลย

นอกจากนี้คำสั่ง **lambda** ยังมีความสามารถสร้างฟังก์ชันขึ้นมาใช้งานได้ครั้งละหลาย ๆ ฟังก์ชัน โดยเป็นการสร้างฟังก์ชัน **lambda** ซ้อนเข้าไปในชนิดข้อมูลลิสต์อีกครั้ง และหากต้องการให้ฟังก์ชันใดทำการประมวลผล ให้ระบุตำแหน่ง (index) ของฟังก์ชันนั้น ๆ ดังตัวอย่างต่อไปนี้

ตัวอย่าง 7.14
การสร้างฟังก์ชัน lambda ครั้งละหลาย ๆ ฟังก์ชันที่อยู่ภายในชนิดข้อมูลลิสต์

```
area_volume = [
        lambda r, h: 3.14 * r * r * h,
        lambda r: 3.14 * r * r,
        lambda b, h: 0.5 * b * h,
        lambda w, h: w * h
]

print('ปริมาตรทรงกระบอก =', area_volume[0](5, 6), 'ลูกบาศก์เมตร')
print('พื้นที่วงกลม = ', area_volume[1](2), 'ตารางเมตร')
print('พื้นที่สามเหลี่ยม = ', area_volume[2](5, 7), 'ตารางเมตร')
print('พื้นที่สี่เหลี่ยมผืนผ้า = ', area_volume[3](10, 12), 'ตารางเมตร')
```

```
ปริมาตรทรงกระบอก = 471.0 ลูกบาศก์เมตร
พื้นที่วงกลม =  12.56 ตารางเมตร
พื้นที่สามเหลี่ยม =  17.5 ตารางเมตร
พื้นที่สี่เหลี่ยมผืนผ้า =  120 ตารางเมตร
```

จากตัวอย่างโปรแกรมได้แสดงให้เห็นถึงการสร้างฟังก์ชัน **lambda** ขึ้นมาใช้งาน ที่มีหลายฟังก์ชันในชนิดข้อมูลลิสต์ ซึ่งมีความสะดวกและประหยัดเวลามากกว่าการใช้คำสั่ง **def** สร้างฟังก์ชัน

ตัวอย่าง 7.15
การสร้างฟังก์ชัน lambda ครั้งละหลาย ๆ ฟังก์ชันที่อยู่ภายในชนิดข้อมูลดิคชันนารี

```
geometry = {
        'cylinderVol': lambda r, h: 3.14 * r * r * h,
        'circleArea': lambda r: 3.14 * r * r,
        'triangleArea': lambda b, h: 0.5 * b * h,
        'rectangleArea': lambda w, h: w * h
}

print('ปริมาตรทรงกระบอก =', geometry['cylinderVol'](5, 6), 'ลูกบาศก์เมตร')
print('พื้นที่วงกลม = ', geometry['circleArea'](2), 'ตารางเมตร')
print('พื้นที่สามเหลี่ยม = ', geometry['triangleArea'](5, 7), 'ตารางเมตร')
print('พื้นที่สี่เหลี่ยมผืนผ้า = ', geometry['rectangleArea'](10, 12), 'ตารางเมตร')
```

```
ปริมาตรทรงกระบอก = 471.0 ลูกบาศก์เมตร
พื้นที่วงกลม =  12.56 ตารางเมตร
พื้นที่สามเหลี่ยม =  17.5 ตารางเมตร
พื้นที่สี่เหลี่ยมผืนผ้า =  120 ตารางเมตร
```

จากตัวอย่างโปรแกรมได้แสดงให้เห็นถึงการสร้างฟังก์ชัน **lambda** ขึ้นมาใช้งาน ที่มีหลายฟังก์ชันในชนิดข้อมูลดิคชันนารี ซึ่งมีความสะดวกกว่าการใช้ข้อมูลลิสต์ในกรณีที่เราไม่สะดวกในการอ้างอิงดัชนี (Index)

7.5 ขอบเขตการเรียกใช้งานตัวแปร

ในหัวข้อนี้เราจะได้เรียนรู้ขอบเขตการเรียกใช้งานตัวแปร (Variable Scope) ในภาษาไพธอน โดยทั่วไปแล้วตัวแปรจะถูกสร้างขึ้นมาสำหรับเก็บข้อมูลใดชนิดหนึ่ง จากนั้นนำไปประมวลผลหาผลลัพธ์ตามที่ผู้พัฒนาโปรแกรมต้องการ ตัวแปรแบ่งออกเป็น 2 ชนิดได้แก่

1. ตัวแปรโกลบอล (Global) คือ ตัวแปรทั่วไปที่สร้างขึ้นมาแล้วเรียกใช้งานได้จากทุกส่วนของโปรแกรม

2. ตัวแปรโลคอล (Local) คือตัวแปรที่สร้างขึ้นมาแล้วเรียกใช้งานได้เฉพาะภายในฟังก์ชันเท่านั้น

เมื่อต้องการสร้างตัวแปรภายในฟังก์ชันแล้วให้ฟังก์ชันอื่น ๆ เรียกใช้งานได้ ให้ใช้คีย์เวิร์ด `global` นำหน้าชื่อตัวแปรนั้น เช่น `global area, global weight` เป็นต้น

ตัวอย่าง 7.16
การสร้างตัวแปรโกลบอลและการเรียกใช้งานโดยฟังก์ชัน

```
def cylinder():
    volume = 3.14 * r * r * h # volume เป็นตัวแปรชนิดข้อมูลโลคอล
    return volume

r = int(input('ป้อนค่ารัศมี : ')) # ตัวแปรชนิดข้อมูลโกลบอล
h = int(input('ป้อนค่าความสูง : ')) # ตัวแปรชนิดข้อมูลโกลบอล
print('ปริมาตรทรงกระบอก = ', cylinder(), 'ลูกบาศก์เมตร')
```

```
ป้อนค่ารัศมี : 2
ป้อนค่าความสูง : 3
ปริมาตรทรงกระบอก =  37.68 ลูกบาศก์เมตร
```

จากตัวอย่างโปรแกรมตัวแปร `r` และ `h` ถูกประกาศไว้เป็นชนิดข้อมูลโกลบอล ทำให้ฟังก์ชันสามารถเรียกใช้งานได้ แต่สำหรับตัวแปร `volume` เป็นตัวแปรชนิดโลคอล ซึ่งจะทำงานได้เฉพาะภายในฟังก์ชันเท่านั้น

ตัวอย่าง 7.17
การสร้างตัวแปรโลคอลขึ้นมาใช้งานภายในฟังก์ชัน และการเรียกใช้งานตัวแปรข้ามฟังก์ชัน

```python
global pi # ตัวแปรชนิดข้อมูลโกลบอล
pi = 3.14

def circle_area():
    global r # ตัวแปรชนิดข้อมูลโกลบอล
    r = 4
    result = pi * (r * r) # ตัวแปรชนิดข้อมูลโลคอล
    return result

def circumference():
    result = 2 * pi * r # ตัวแปรชนิดข้อมูลโลคอล
    return result

print(f'พื้นที่วงกลม = {circle_area()} ตารางเมตร')
print(f'เส้นรอบวงกลม = {circumference()} เมตร')
```

```
พื้นที่วงกลม = 50.24 ตารางเมตร
เส้นรอบวงกลม = 25.12 เมตร
```

จากตัวอย่างโปรแกรมได้ประกาศตัวแปร `r` และ `pi` ให้เป็นตัวแปรโกลบอล จึงมีผลทำให้ฟังก์ชัน `circumference()` สามารถเรียกใช้งานและนำค่าตัวแปรไปคำนวณหาเส้นรอบวงกลมได้ทั้ง ๆ ที่ไม่ได้กำหนดค่าในโปรแกรมหลัก

ทั้งนี้การประกาศตัวแปรโกลบอลจะทำภายใต้ขอบเขตใดของโปรแกรมก็ได้ ในตัวอย่างนี้เราประกาศ `pi` ให้เป็นตัวแปรโกลบอลนอกขอบเขตของฟังก์ชัน และได้ประกาศ `r` ให้เป็นตัวแปรโกลบอลในขอบเขตของฟังก์ชัน แต่เราก็สามารถเรียกใช้งานตัวแปรโกลบอลทั้งสองตัวนี้ใน `circumference()` ได้

ตัวอย่าง 7.18
การสร้างตัวแปรโกลบอลและโลคอลที่มีชื่อตัวแปรเหมือนกัน

```
1  def calculate(cash):
2      total = cash - (cash*vat) # ตัวแปรชนิดข้อมูลโลคอล
3      return total
4
5  cash = float(input('ป้อนจำนวนเงิน : '))
6  vat = 0.07 # ตัวแปรชนิดข้อมูลโกลบอล
7  total = cash - (cash*vat) # ตัวแปรชนิดข้อมูลโกลบอล
8
9  print(f'จำนวนเงินไม่รวมภาษี 7% (ตัวแปรแบบโลคอล) = {calculate(100)} บาท')
10 print(f'จำนวนเงินไม่รวมภาษี 7% (ตัวแปรแบบโกลบอล) = {total} บาท')
```

```
ป้อนจำนวนเงิน : 1000
จำนวนเงินไม่รวมภาษี 7% (ตัวแปรแบบโลคอล) = 93.0 บาท
จำนวนเงินไม่รวมภาษี 7% (ตัวแปรแบบโกลบอล) = 930.0 บาท
```

จากตัวอย่าง 7.18 มีการสร้างตัวแปร `total` ไว้สองตำแหน่งคือ บรรทัดที่ 3 และบรรทัดที่ 5 ซึ่งอยู่ภายในฟังก์ชัน `calculate()` เมื่อนำค่าตัวแปรมาแสดงผลจะได้ผลลัพธ์ไม่เหมือนกัน เนื่องจากค่าตัวแปร `total` ในบรรทัดที่ 5 จะทำงานเฉพาะภายในฟังก์ชัน `calculate()` เท่านั้น โดยรับค่าจากบรรทัดที่ 7 ไปประมวลผล ส่วนค่าตัวแปร `vat` เป็นตัวแปรโกลบอล ทำให้ฟังก์ชัน `calculate()` เรียกใช้งานและนำไปประมวลผลได้ ซึ่งตัวแปรแบบโลคอลจะนำ **`100`** ไปคำนวณ (บรรทัดที่ 7) ส่วนตัวแปรแบบโกลบอลจะนำ **`1000`** ไปคำนวณ

ตัวอย่าง 7.19
การสร้างตัวแปรโลคอลที่มีชื่อตัวแปรเหมือนกันเพื่อใช้งานในฟังก์ชัน

```
def sal_rate_1(m, ot):
    ot_rate = 0
    bonus = 0.5
    total =  (m + (ot*ot_rate)) * bonus
    salary = total + m
    return salary

def sal_rate_2(m, ot):
    ot_rate = 20
    bonus = 0.2
    total =  (m + (ot*ot_rate)) * bonus
    salary = total + m
    return salary

print('เงินเดือนที่ได้รับสุทธิ = ', sal_rate_1(12000, 5), 'บาท')
print('เงินเดือนที่ได้รับสุทธิ = ', sal_rate_2(20000, 5), 'บาท')
```

```
เงินเดือนที่ได้รับสุทธิ =  18000.0 บาท
เงินเดือนที่ได้รับสุทธิ =  24020.0 บาท
```

จากตัวอย่างโปรแกรมได้สร้างฟังก์ชันขึ้นมาใช้งาน 2 ฟังก์ชัน สำหรับคำนวณเงินเดือนสุทธิ หลังจากมีการคิดค่าโบนัสและค่าทำงานล่วงเวลา ให้เราสังเกตเห็นว่าทั้งสองฟังก์ชันนี้มีตัวแปรเหมือนกัน ซึ่งตัวแปรจะทำงานแบบตัวแปรโลคอล คือ ดำเนินการเฉพาะภายในฟังก์ชันเท่านั้น และไม่สามารถเรียกใช้งานข้ามฟังก์ชันได้

7.6 ฟังก์ชัน Built-in ภาษาไพธอน

ภาษาไพธอนได้จัดเตรียมฟังก์ชันต่าง ๆ ไว้ให้ผู้พัฒนาโปรแกรมเรียกใช้งานเป็นจำนวนมาก `built-in function` เป็นประเภทฟังก์ชันที่สามารถเรียกใช้งานได้เลย โดยไม่ต้องใช้คำสั่ง `import` โมดูลใด ๆ เข้ามาในโปรแกรมก่อน และจากหลาย ๆ บทที่ผ่านมาได้เรียกใช้งาน

กันไปบ้างแล้ว ในส่วนนี้จะสรุปและนำเสนอ `built-in function` อื่น ๆ เพิ่มเติม ที่ผู้อ่านอาจจะได้นำมาใช้งานในการพัฒนาโปรแกรม

เมธอด	ความหมาย	รูปแบบการใช้งาน	คำอธิบายเพิ่มเติม
`abs()`	ใช้สำหรับหาค่าจำนวนเต็มบวก (absolute)	`abs(x)`	`x` คือ ชนิดข้อมูลจำนวนเต็มหรือทศนิยม
`all()`	ใช้สำหรับตรวจสอบสมาชิกทุกตัวเป็นค่าจริง (True) หรือไม่	`all(seq)`	`seq` คือ ตัวแปรหรือชนิดข้อมูลแบบลำดับทั้งลิสต์และทูเพิล
`any()`	ใช้สำหรับตรวจสอบตัวแปรชนิดข้อมูลแบบเรียงลำดับว่ามีค่าว่างหรือไม่ ถ้าตัวแปรเก็บค่าว่างจะคืนค่ากลับ `False` ถ้ามีสมาชิกอยู่จะคืนค่า `True`	`any(seq)`	`seq` คือ ตัวแปรหรือชนิดข้อมูลแบบลำดับทั้งลิสต์และทูเพิล
`ascii()`	ใช้สำหรับแปลงสัญลักษณ์เป็นรหัสแอสกี	`ascii(obj)`	`obj` คือ ชนิดข้อมูลอักขระหรือสตริง ลิสต์ ทูเพิล เป็นต้น
`callable()`	ใช้สำหรับตรวจสอบออบเจ็คว่าเรียกใช้งานได้หรือไม่ ส่งค่ากลับเป็นค่า `True` ถ้าเรียกใช้งานได้ และคืนค่า `False` เมื่อไม่สามารถเรียกใช้งานได้	`callable(obj)`	`obj` คือ ออบเจ็คที่ต้องการตรวจสอบ

เมธอด	ความหมาย	รูปแบบการใช้งาน	คำอธิบายเพิ่มเติม
chr()	ใช้สำหรับแปลงรหัส Unicode ให้เป็นตัวอักขระ	chr(i)	i คือ รหัส Unicode
compile()	ใช้สำหรับคอมไพล์คำสั่งภาษาไพธอนหรือซอร์สโค้ด ส่งค่ากลับคืนมาเป็นออบเจ็ค code มีการใช้งานเหมือนกับฟังก์ชัน exec() และ eval()	compile(source, filename, mode)	source คือ สตริงหรือไบต์สตริง filename คือ แหล่งที่เก็บไฟล์หรือไฟล์ที่ต้องอ่านขึ้นมาคอมไพล์ mode คือ มีด้วยกัน 3 โหมด eval ใช้กับซอร์สโค้ดที่มีนิพจน์เดียวเท่านั้น exec ใช้กับซอร์สโค้ดที่ประกอบด้วยฟังก์ชันหรือคลาสได้ single ใช้กับซอร์สโค้ดที่มีการโต้ตอบกับผู้ใช้งาน
divmod()	จะคืนค่าผลหารและเศษจากการหารออกมา	divmod(x, y)	x คือ ตัวตั้ง y คือ ตัวหาร
enumerate()	ใช้สำหรับกำหนดหมายเลขแบบเรียงลำดับ	enumerate(seq [, start])	seq คือ ชนิดข้อมูลแบบเรียงลำดับ start คือ ค่าเริ่มต้น
eval()	ใช้สำหรับประมวลผลคำสั่งนิพจน์ที่อยู่ในรูปแบบชนิดข้อมูลอักขระหรือสตริง	eval(expression	expression คือ นิพจน์ที่อยู่ในรูปแบบชนิดข้อมูลอักขระหรือสตริง

เมธอด	ความหมาย	รูปแบบการใช้งาน	คำอธิบายเพิ่มเติม
`exec()`	ใช้สำหรับ execute คำสั่งโปรแกรม	`exec(obj[, globals, locals])`	`obj` คือ ข้อมูลอักขระ, สตริง หรือคำสั่งโปรแกรม `globals`, `locals`
`filter()`	ใช้สำหรับกรองข้อมูลจากชนิดข้อมูลแบบเรียงลำดับ	`filter(func, seq)`	`func` คือ ฟังก์ชันที่ทำหน้าที่กรองข้อมูลจากชนิดข้อมูลแบบเรียงลำดับ `seq` คือ ชนิดข้อมูลแบบเรียงลำดับ
`getattr()`	ใช้สำหรับเรียกดูค่าข้อมูลแอตทริบิวต์ในออบเจ็ค	`getattr (obj, name[, default])`	`obj` คือออบเจ็คที่ถูกเรียกดูค่าแอตทริบิวต์ `name` คือ ชื่อแอตทริบิวต์ `default` คือ กำหนดข้อความแสดงผล
`hasattr()`	ใช้สำหรับตรวจสอบชื่อแอตทริบิวต์มีอยู่ในออบเจ็คหรือไม่ ถ้ามีคืนค่า **`True`** ถ้าไม่มีคืนค่า **`False`**	`hasattr(obj, name)`	`obj` คือ ออบเจ็คที่ถูกดูชื่อแอตทริบิวต์ `name` คือ ชื่อแอตทริบิวต์ที่ต้องการตรวจสอบมีอยู่ในออบเจ็คหรือไม่
`help()`	ใช้สำหรับกรณีที่ต้องการขอความช่วยเหลือวิธีเรียกใช้คำสั่งต่าง ๆ	`help(object)`	`obj` คือ ออบเจ็คที่ต้องการวิธีการใช้งาน
`map()`	ใช้สำหรับหาผลลัพธ์จากชนิดข้อมูลแบบเรียงลำดับจากฟังก์ชันที่กำหนด	`map(func, seq1, ... seq_n)`	`func` คือ ฟังก์ชันที่สร้างขึ้นมาหาผลลัพธ์ `seq1,...seq_n` คือ ชนิดข้อมูลแบบเรียงลำดับ

เมธอด	ความหมาย	รูปแบบการใช้งาน	คำอธิบายเพิ่มเติม
`ord()`	ใช้สำหรับแปลงตัวอักขระเป็นค่ารหัส Unicode	`ord(s)`	`s` คือ ตัวอักขระที่ต้องการแปลงเป็นรหัส Unicode
`range()`	ใช้สำหรับสร้างกลุ่มข้อมูลตัวเลขจำนวนเต็ม	`range(start, stop [, step])`	`start` คือ ค่าเริ่มต้น `stop` คือ ค่าถัดจากค่าสุดท้ายใน `range` นั้น `step` คือ ค่าที่ให้เพิ่มขึ้นแต่ละชั้น
`repr()`	ใช้สำหรับแสดงชนิดข้อมูลอักขระหรือสตริงที่มีเครื่องหมาย Single Quote (`'...'`) ถ้าใช้ฟังก์ชัน `print()` แสดงผล ถ้าไม่ใช้จะมีเครื่องหมาย Double Quote (`"..."`)	`repr(obj)`	`obj` คือ ชนิดข้อมูลที่ต้องการแสดงผล
`reversed()`	ใช้สำหรับแสดงชนิดข้อมูลแบบเรียงลำดับแบบย้อนกลับ	`reversed(seq)`	`seq` คือ ชนิดข้อมูลแบบเรียงลำดับ
`round()`	ใช้สำหรับปัดเศษจำนวนทศนิยม	`round(number [, ndigits)`	`number` คือตัวเลขที่ต้องการปัดเศษ `ndigits` คือจำนวนทศนิยมที่ต้องการ ถ้าไม่กำหนดจะไม่มีทศนิยม

เมธอด	ความหมาย	รูปแบบการใช้งาน	คำอธิบายเพิ่มเติม
`slice()`	ใช้สำหรับแสดงค่าข้อมูลตามช่วงที่กำหนด	`slice(start, stop, step)`	`start` คือ จุดเริ่มต้นที่ต้องการตัด `stop` คือ จุดสุดท้ายที่ต้องการตัด `step` คือ ค่าช่วงห่างในการตัด
`sorted()`	ใช้สำหรับจัดเรียงข้อมูลในชนิดข้อมูลแบบเรียงลำดับ คืนค่ากลับมาเป็นชนิดข้อมูลลิสต์	`sorted(seq [, key][, reverse])`	`seq` คือ ชนิดข้อมูลแบบเรียงลำดับ `key` คือ วิธีการจัดเรียงข้อมูลค่าปกติเรียงจากน้อยสุดไปหามากสุด `reverse` คือ ถ้ากำหนดเป็น `True` จะเรียงจากค่ามากไปหาน้อย
`vars()`	ใช้สำหรับดูค่าข้อมูลที่ถูกเก็บไว้ในตัวแปร	`vars(obj)`	`obj` คือ ออบเจ็ค โมดูล คลาสหรืออินสแตนซ์ (instance) ที่มีการเก็บค่าข้อมูลในตัวแปร
`zip()`	ใช้สำหรับรวมชนิดข้อมูลแบบเรียงลำดับเข้าด้วยกัน คืนค่ากลับมาเป็นชนิดข้อมูลทูเพิล	`zip(*seq)`	`seq` คือ ชนิดข้อมูลแบบเรียงลำดับ

ในลำดับถัดไปเป็นตัวอย่างการใช้งานบางฟังก์ชัน `Built-in` ของภาษาไพธอน เพื่อเป็นแนวทางนำไปประยุกต์ใช้งาน สำหรับฟังก์ชันที่ผู้เขียนไม่ได้นำมาแสดงเป็นตัวอย่างการใช้งานนั้น ผู้อ่านสามารถทดสอบและทำความเข้าใจผลลัพธ์โดยใช้ตัวอย่างเป็นแนวทาง

ตัวอย่าง 7.20
การใช้งานฟังก์ชัน `abs()`, `all()`, `any()`, `ascii()` และ `chr()`

```
x = -10.5
y = [1, 2, 3, True, False]
z = []
msg = 'Python programming'
print(abs(x))
print(all(y))
print(any(z))
print(ascii(msg))
print('ตัวอักขระรหัส 97 =', chr(97))
```

```
10.5
False
False
'Python programming'
ตัวอักขระรหัส 97 = a
```

ตัวอย่าง 7.21
การใช้งานฟังก์ชัน divmod(), eval(), ord(), reversed() และ zip()

```
l = [1, 2, 3, 4, 5]
s = {'Python', 'Java','C', 'C++', 'R'}
sl = 'Python Programming'
div = divmod(9, 2)
exp = '20 + 60'
print('ค่าผลหารและเศษ = ', div)
print('exp =', eval(exp))
print('รหัส Unicode A =', ord('A'))
print(list(reversed(l)))
print('ค่าข้อมูลตำแหน่งที่ 1-10 ข้ามครั้งละ 2 ตำแหน่ง = ',
        sl[slice(1, 10, 2)])
print(dict(zip(l, s)))
```

```
ค่าผลหารและเศษ = (4, 1)
exp = 80
รหัส Unicode A = 65
[5, 4, 3, 2, 1]
ค่าข้อมูลตำแหน่งที่ 1-10 ข้ามครั้งละ 2 ตำแหน่ง =  yhnPo
{1: 'Python', 2: 'C++', 3: 'C', 4: 'Java', 5: 'R'}
```

สรุปท้ายบท

ฟังก์ชันเป็นส่วนหนึ่งที่มีความสำคัญในการพัฒนาโปรแกรม ซึ่งในบทนี้ได้แนะนำวิธีการสร้างฟังก์ชันขึ้นมาใช้งานแบบต่าง ๆ เช่น การสร้างฟังก์ชันที่ไม่มีการส่งค่า การสร้างฟังก์ชันที่ไม่มีการคืนค่า การสร้างฟังก์ชันที่มีการรับค่าแต่ไม่มีการส่งค่ากลับ เป็นต้น นอกจากนี้ยังได้รู้จักกับวิธีการส่งค่าอาร์กิวเมนต์ให้กับค่าพารามิเตอร์และยังได้รู้จักการวิธีการสร้างค่าพารามิเตอร์ไว้รอรับค่าอาร์กิวเมนต์ รวมไปถึงสร้างฟังก์ชันที่ไม่ต้องกำหนดชื่อด้วยคำสั่ง **lambda** ทำให้ลด

เวลาในการเขียนคำสั่งโปรแกรม อีกทั้งยังรู้จักกับฟังก์ชัน `Built-in` ของภาษาไพธอน ซึ่งเรียกใช้งานโดยไม่ต้องใช้คำสั่ง `import`

แบบฝึกหัด

1. จงเขียนคำสั่งโปรแกรมคำนวณหาปริมาตรพีระมิด (Pyramid), ทรงกระบอก (Cylinder) และทรงกรวย (Cone) โดยให้แยกการคำนวณแต่ละปริมาตรออกเป็นฟังก์ชัน โดยมีเมนูแสดงให้ผู้ใช้งานเลือกการหาปริมาตรที่ต้องการได้ กำหนดให้ป้อนข้อมูลผ่านทางคีย์บอร์ด

2. จงเขียนโปรแกรมแบบฟังก์ชันคำนวณอัตราแลกเปลี่ยนเงินไทย เป็นสกุลเงินในกลุ่มประเทศอาเซียน โดยมีเมนูแสดงให้ผู้ใช้งานเลือกสกุลเงินที่ต้องการเปลี่ยน และให้ผู้ใช้งานป้อนจำนวนเงินผ่านทางคีย์บอร์ด

3. จงเขียนโปรแกรมคำนวณหาค่าดัชนีมวลกาย โดยแยกการคำนวณออกเป็นฟังก์ชันระหว่างเพศชายและเพศหญิง กำหนดให้ผู้ใช้งานต้องป้อนข้อมูลผ่านทางคีย์บอร์ด