**สรุปเนื้อหาเรื่อง Array**

อาเรย์(array) คือการเก็บข้อมูลเป็นชุดของข้อมูลที่มีลำดับของข้อมูล โดยจะประกอบด้วยข้อมูลหลายๆข้อมูลและมีชนิดของข้อมูลแบบเดียวกัน เช่น อาเรย์ของเลขจำนวนเต็ม(int), อาเรย์ของเลขทศนิยม(float), อาเรย์ของข้อความ(string) ซึ่งโดยปกติจะมีความแตกต่างจากชุดข้อมูลชนิดลิสต์ที่สามารถเก็บข้อมูลได้หลายชนิดภายในลิสต์เดียวกัน ใน python นั้นโดยปกติแล้วจะไม่มีชุดข้อมูลชนิดอาเรย์(array)มาให้ แต่หากต้องการใช้งานในรูปแบบอาเรย์ก็จะใช้การเก็บชุดข้อมูลชนิดลิสต์(list) แต่จะทำการเก็บข้อมูลเพียงชนิดเดียวภายในลิสต์ดังกล่าวแทน เพื่อนำไปใช้งานในรูปแบบของอาเรย์(array) ในกรณีที่ต้องใช้งานรูปแบบการเก็บข้อมูลชนิดอาเรย์โดยเฉพาะ ก็จำเป็นต้องใช้ไลบรารี่(library)เพิ่มเติม เช่น ชุดข้อมูลชนิดอาเรย์จากไลบรารี่ numpy แทน ซึ่งหากต้องการดำเนินการทางคณิตศาสตร์เฉพาะเจาะจงกับชุดข้อมูลชนิดอาเรย์นั้น การเลือกใช้งานอาเรย์จากไลบรารี่ numpy แทนการใช้การเก็บข้อมูลในรูปแบบลิสต์จะทำให้มีประสิทธิภาพมากกว่า

การเลือกใช้ชุดข้อมูลชนิดลิสต์(List) มาแทนเพื่อใช้งานในรูปแบบอาเรย์(Array)โดยตรงนั้นจะมีข้อแตกต่างจากการใช้อาเรย์(Array) ปกติ เพราะชุดข้อมูลชนิดลิสต์(List) สามารถจะเก็บข้อมูลต่างชนิดกันได้ โดยเราสามารถจะเก็บได้ทุกข้อมูลเช่น เลขจำนวนเต็ม(integer), เลขทศนิยม(float) และ ข้อความ(string) อยู่ภายในลิสต์เดียวกัน ดังนั้นรูปแบบของการเก็บข้อมูลนั้นจะต้องซับซ้อนขึ้นเพื่อรองรับการทำงานในลักษณะดังกล่าว ทำให้ประสิทธิภาพในการประมวลผลในบางกรณีอาจจะลดลงมากจนสังเกตเห็นได้ชัดเจน

**การใช้งาน อาเรย์ (Array) เบื้องต้น**

numpy เป็นมอดูลที่ทำให้เราสามารถใช้ออบเจ็กต์ชนิดที่เรียกว่า ndarray ซึ่งหมายถึงอาเรย์หลายมิติ บางครั้งก็เรียกว่าอาเรย์เฉยๆ คำว่า "อาเรย์" นั้นเป็นคำที่ถูกใช้ในภาษาอื่นๆอีกหลายภาษา แต่ว่าสำหรับในภาษาไพธอนสิ่งที่มีคุณสมบัติเทียบเท่ากับอาเรย์ในภาษาอื่น นั้นกลับเรียกว่าลิสต์ ส่วนอาเรย์จะหมายถึง ndarray ของ numpy ก่อนที่จะเริ่มใช้งานสิ่งที่ต้องทำเป็นอย่างแรกก็คือทำการ import เรียกใช้ขึ้นมาก่อน

import numpy as np

ตัวย่อ np นี้จะใช้แบบนี้ไปตลอด เพราะค่อนข้างเป็นสากล แม้แต่เวลาที่เรียกชื่อฟังก์ชันต่างๆใน numpy ก็จะเรียกโดยขึ้นต้นด้วย np.

### การสร้างอาเรย์ขึ้นจากลิสต์

มีหลายวิธีแต่วิธีที่พื้นฐานที่สุดคือสร้างขึ้นมาจากลิสต์, ทูเพิล หรือเรนจ์ โดยใช้ np.array(ลิสต์)

aray = np.array(range(3,7)) (ได้ผลลัพธ์ [ 3 4 5 6 ]

araya = np.array([[1,2],[3,4]]) (ได้ผลลัพธ์ [[1 2 ]

[3 4 ]])

print(aray)

print(araya)

จะเห็นว่าไม่ต่างอะไรจากลิสต์นัก แต่ความจริงแล้วต่างไปพอสมควร อย่างแรกก็คือเวลาที่เราสั่ง print ผลลัพธ์จะออกมาเป็นแถวเป็นระเบียบอย่างที่เห็นโดยอัตโนมัติ ทำให้ดูเข้าใจง่ายขึ้นมากต่อมาคืออาเรย์มีแอตทริบิวต์ติดตัวที่สามารถให้ข้อมูลของตัวอาเรย์นั้น เช่น

**shape** รูปร่างของอาเรย์

**size** จำนวนสมาชิกในอาเรย์

**ndim** จำนวนมิติของอาเรย์

ลองเอาอาเรย์ ๒ ตัวจากตัวอย่างเมื่อครู่มาหาแอตทริบิวต์

print(aray.shape) # ได้ (4,) ผลลัพธ์ (4,)

print(aray.size) # ได้ 4 4

print(aray.ndim) # ได้ 1 1

print(araya.shape) # ได้ (2, 2) (2, 2)

print(araya.size) # ได้ 4 4

print(araya.ndim) # ได้ 2 2

จะเห็นว่าลิสต์ที่ใช้นั้นมีการซ้อนกันก็จะได้เป็นอาเรย์ 2 มิติ โดยมีแนวตั้งเป็นมิติที่หนึ่ง แนวนอนเป็นมิติที่สองและยังสามารถซ้อนกันเป็นมิติที่สูงขึ้นไปอีกได้ เช่นลองสร้าง 3 มิติ

araye = np.array([[[1,2],[3,4]],[[5,6],[7,8]]]) ผลลัพธ์คือ [[[1 2]

print(araye) [3 4 ]]

print(araye.shape) [[5 6 ]

print(araye.size) [ 7 8 ]]]

print(araye.ndim) (2,2,2)

8

3

**ข้อควรระวังคือลิสต์ที่จะใช้สร้างอาเรย์จะต้องมีจำนวนสมาชิกในลิสต์ย่อยแต่ละลิสต์เท่ากันไม่เช่นนั้นจะถูกตีความเป็นออบเจ็กต์ทั่วไป**

**ชนิดของข้อมูลในอาเรย์**

สามารถตรวจสอบชนิดของข้อมูลในอาเรย์ได้โดยดูที่แอตทริบิวต์ชื่อ dtype  
 araye = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])

arayu = np.array([[1,2],[3,4,5]])

print(araye.dtype) # ได้ int64

print(arayu.dtype) # ได้ object

จะเห็นว่า arayu ซึ่งสร้างขึ้นมาจากลิสต์ที่มีจำนวนสมาชิกไม่เท่ากันนั้นได้ข้อมูลประเภท object แทนที่จะเป็น int อย่างที่ควรเป็น ส่วน araye นั้นกลายเป็นชนิด int ตามที่ควรจะเป็น โดยเลข 64 ใน int64 นี้บอกถึงขนาดของหน่วยความจำเป็นบิตที่ใช้ในการเก็บตัวเลข ปกติแล้วในภาษาไพธอนจำนวนเต็มจะไม่ได้แบ่งชนิดย่อย แต่ในบางภาษาเช่นภาษาซีจำนวนเต็มจะถูกแบ่งเป็นชนิดตามขนาดของหน่วยความจำที่ ใช้

จะเห็นว่าข้อมูลในอาเรย์จะต้องประกอบจากตัวแปรชนิดเดียวกันหมดทุกตัว หากตอนที่สร้างอาเรย์ขึ้นจากลิสต์นั้นมีสมาชิกที่ชนิดต่างกันจะถูกทำให้เหมือนกันหมด

กรณีที่มีสายอักขระปนอยู่ ตัวอื่นก็จะถูกเปลี่ยนเป็นสายอักขระไปด้วย เช่น

arayo = np.array([[1,2],[3.,'4']])

print(arayo)

print(arayo.dtype)

[['1' '2']

['3.0' '4']]

<U32

U ในที่นี้หมายถึงเป็นยูนิโค้ด และ 32 เป็นจำนวนหน่วยความจำที่ใช้ โดยปกติจะเท่ากับจำนวนตัวอักษรแต่ถ้าสั้นกว่า 32 จะถูกกำหนดให้เป็น 32

ความยาวของสายอักขระสามารถกำหนดขึ้นเองได้ และถ้ากำหนดความยาวต่ำกว่าจำนวนตัวอักษรก็จะถูกตัดทอนหายไป

อาเรย์สามารถเก็บข้อมูลชนิดสายอักขระได้ก็จริง แต่โดยทั่วไปมักจะใช้กับจำนวนตัวเลขมากกว่า เพื่อที่จะใช้ประโยชน์ในเรื่องการคำนวณอย่างเต็มที่

ชนิดของอาเรย์ทั้งหมดสามารถดูได้ที่แอตทริบิวต์ np.sctypes

ชนิดของสมาชิกในอาเรย์สามารถเปลี่ยนได้ด้วยเมธอด astype บนตัวอาเรย์

### การอ้างอิงถึงข้อมูลในอาเรย์

เช่น เดียวกับลิสต์ อาเรย์ก็ใช้การเติมวงเล็บเหลี่ยม [ ] เพื่ออ้างอิงข้อมูล โดยเลขในกรณีสองมิตินั้น ตัวแรกคือดัชนีของแนวตั้ง (เลขแถว) และตัวหลังคือแนวนอน (เลขหลัก)

ariyu = np.array([[1,2,3],[4,5,6]])

print(ariyu[0][1]) # ได้ 2

print(ariyu[1][2]) # ได้ 6

การเข้าถึงสมาชิกทีละหลายตัวก็ทำได้ด้วยการใช้โคลอน : เช่นเดียวกับลิสต์ แต่สำหรันอาเรย์สองมิติแทนที่จะใช้วงเล็บเหลี่ยมวางต่อกันสองอันสามารถใช้จุลภาคแทนได้เช่นกัน

print(ariyu[1][:]) # ได้ [4 5 6]

print(ariyu[1,:]) # ได้ [4 5 6]

สรุปเป็นตารางเพื่อให้เห็นภาพชัด สมมุติว่าอาเรย์มี n แถว m หลัก

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| แถว/หลัก | 0 | 1 | 2 | 3 | .. | m-1 |
| 0 | [0,0] | [0,1] | [0,2] | [0,3] | [0,..] | [0,m-1] |
| 1 | [1,0] | [1,1] | [1,2] | [1,3] | [1,..] | [1,m-1] |
| 2 | [2,0] | [2,1] | [2,2] | [2,3] | [2,..] | [2,m-1] |
| 3 | [3,0] | [3,1] | [3,2] | [3,3] | [3,..] | [3,m-1] |
| .. | [..,0] | [..,1] | [..,2] | [..,3] | [..,..] | [..,m-1] |
| n-1 | [n-1,0] | [n-1,1] | [n-1,2] | [n-1,3] | [n-1,..] | [n-1,m-1] |

### การเขียนทับข้อมูลในอาเรย์

เช่นเดียวกับลิสต์ อาเรย์ก็สามารถเขียนทับแก้ข้อมูลข้างในเมื่อไหร่ก็ได้ด้วยการใส่ค่าทับลงไป

aruy = np.array([[2,2,2],[2,2,2],[2,2,2]])

print(aruy)

aruy[1,1] = 3

print(aruy)

[[2 2 2]

[2 2 2]

[2 2 2]]

[[2 2 2]

[2 3 2]

[2 2 2]]

ถ้าเขียน [:,:] จะเป็นการอ้างถึงข้อมูลทั้งหมด และทุกตัวจะถูกเปลี่ยนให้มีค่าเท่ากันหมดลักษณะนี้เรียกว่าเป็นการกระจายค่า (broadcast) คือสามารถใช้เลขตัวเดียวมาแทนค่าเพื่อยัดใส่แทนสมาชิกหลายตัวในอาเรย์ได้ในทีเดียว

**การสร้างอาเรย์ด้วยฟังก์ชัน**  
นอกจากจะสร้างจากลิสต์แล้วอาเรย์ก็ยังสร้างจากฟังก์ชันต่างๆที่มีอยู่ในมอดูล numpy เองได้  
  
ฟังก์ชันที่ใช้บ่อย ได้แก่

|  |  |
| --- | --- |
| np.arange | สร้างอาเรย์หนึ่งมิติที่มีเลขเรียงกัน |
| np.linspace | สร้างอาเรย์หนึ่งมิติตามจำนวนที่กำหนดโดยเว้นช่วงเท่าๆกัน |
| np.ones | สร้างอาเรย์ที่มีแต่เลข 1 ตามขนาดที่กำหนด |
| np.zeros | สร้างอาเรย์ที่มีแต่เลข 0 ตามขนาดที่กำหนด |
| np.full | สร้างอาเรย์ที่มีแต่เลขอะไรก็ได้เลขเดียวตามที่กำหนด โดยมีขนาดตามที่กำหนด |
| np.empty | สร้างอาเรย์เปล่าๆ ตามขนาดที่กำหนด |
| np.identity | สร้างอาเรย์ที่เป็นเมทริกซ์เอกลักษณ์ (เมทริกซ์สองมิติที่มีค่าเป็น 1 เมื่อพิกัดแนวตั้งและนอนเท่ากัน นอกนั้นเป็น 0) |
| np.eye | เหมือนกับ np.identity แต่ขนาดแนวตั้งและนอนไม่ต้องเท่ากันก็ได้ |

**คุณสมบัติการถ่ายทอดของอาเรย์**  
อาเรย์ที่ถูกสร้างขึ้นมาถือเป็นออบเจ็กต์อันหนึ่ง ถ้าหากมีตัวแปรมารับด้วย = มันก็จะแทนออบเจ็กต์อาเรย์นั้น และหากมีตัวแปรมารับต่อตัวแปรนั้นด้วย = อีก จะกลายเป็นว่าตัวแปรใหม่นั้นแทนออบเจ็กต์อาเรย์ตัวเดียวกัน หากมีการเปลี่ยนแปลงค่าภายในก็จะมีการเปลี่ยนแปลงไปตามกัน

aime = np.array([1,2,3,4,5])

aiment = aime

print(aiment) # ได้ [1 2 3 4 5]

print(aime) # ได้ [1 2 3 4 5]

aiment[1] = 6

print(aiment) # ได้ [1 6 3 4 5]

print(aime) # ได้ [1 6 3 4 5]

จะเห็นว่า aiment = aime ทำให้ aiment กลายเป็นตัวแปรที่ใช้แทน aime พอมีการเปลี่ยนแปลงภายใน aiment (เช่นในที่นี้คือแก้ค่าสมาชิก [1] เป็น 6) aime ก็จะเปลี่ยนแปลงตาม  
คุณสมบัติข้อนี้สามารถใช้กับลิสต์ได้ผลในทำนองเดียวกัน จึงดูไม่ใช่เรื่องอะไรแปลกใหม่

แต่นอกจากนี้แล้วอาเรย์ยังสามารถถ่ายทอดด้วยชิ้นส่วนที่ถูกตัดมาด้วย

aime = np.array([1,2,3,4,5])

aimez = aime[2:]

print(aimez) # ได้ [3 4 5]

print(aime) # ได้ [1 2 3 4 5]

aimez[1] = 8

print(aimez) # ได้ [3 8 5]

print(aime) # ได้ [1 2 3 8 5]

จะเห็นว่า aimez = aime[2:] ทำให้ aimez กลายเป็นตัวแทนของส่วนตั้งแต่ตำแหน่ง [2] ของ aime จากนั้นพอมีการแก้ค่าภายใน aimez ค่าภายใน aime ก็จะถูกเปลี่ยนแปลงไปด้วย

คุณสมบัติตรงนี้อาจเป็นได้ทั้งประโยชน์และโทษ  
ที่อาจเป็นโทษได้นั้นก็เพราะว่าหากใครรู้เท่าไม่ถึงการแค่อยากจะได้อาเรย์ใหม่ ที่เอาค่าของอาเรย์เก่ามาใช้ แต่กลับกลายเป็นว่าเป็นการสร้างอาเรย์ที่เป็นตัวแทนของส่วนหนึ่งของอาเรย์ เก่าขึ้นมา แบบนี้พอแก้อะไรในอาเรย์ใหม่ อาเรย์เก่าก็จะถูกเปลี่ยนไปด้วย ซึ่งเป็นเหตุการณ์ที่อันตราย  
เพื่อที่จะหลีกเลี่ยงเรื่องแบบนั้น จำเป็นจะต้องใช้เมธอด copy() เวลาที่ต้องการสร้างอาเรย์ใหม่จากอาเรย์เก่า แบบนี้จะเป็นการสร้างอาเรย์ใหม่ขึ้นมาโดยไม่เกี่ยวกับตัวเก่าอีกเลย

aime = np.array([1,2,3,4,5])

aimons = aime[2:].copy()

print(aimons) # ได้ [3 4 5]

print(aime) # ได้ [1 2 3 4 5]

aimons[1] = 8

print(aimons) # ได้ [3 8 5]

print(aime) # ได้ [1 2 3 4 5]

จะเห็นว่าเมื่อเติม .copy() ต่อท้ายลงไปตอนสร้างอาเรย์ aimons ขึ้นมาจะทำให้อาเรย์นี้กลายเป็นอาเรย์ใหม่ที่ไม่มีความเกี่ยวข้องอะไรกับ aime ทีนี้แม้ว่าจะมีการเปลี่ยนแปลงค่าภายในไปก็จะไม่กระทบต่ออาเรย์เก่า  
  
copy นี้นอกจากจะเขียนในรูปเมธอดต่อท้ายแบบนี้ยังอาจเขียนในรูปฟังก์ชัน เป็น np.copy() ได้ด้วย

**สรุปข้อแตกต่างระหว่างอาเรย์กับลิสต์**

* อาเรย์จะต้องประกอบด้วยข้อมูลเพียงชนิดเดียวเท่านั้น
* อาเรย์ต้องมีจำนวนข้อมูลในแต่ละแถวเท่ากันหมด
* อาเรย์สามารถคำนวณทางคณิตศาสตร์ได้โดยตรง
* อาเรย์คำนวณได้เร็วกว่า
* อาเรย์มีวิธีการเข้าถึงข้อมูลภายในได้ยืดหยุ่นกว่า
* อาเรย์มีคุณสมบัติการถ่ายทอดภายในชิ้นส่วนประกอบ

**สรุปเนื้อหาเรื่อง Function**

ฟังก์ชัน (Function) คือส่วนของโค้ดหรือโปรแกรมที่ทำงานเพื่อวัตถุประสงค์บางอย่าง ในภาษา Python เราสามารถสร้างฟังก์ชันของเราเองเพื่อให้ทำงานอย่างที่เราต้องการ ในการเขียนโปรแกรมนั้นเรามักจะแยกโค้ดที่มีการทำงานเหมือนๆ กันเป็นฟังก์ชันเอาไว้ และเรียกใช้ฟังก์ชันนั้นซ้ำๆ ซึ่งเป็นแนวคิดของการ reuse โค้ด

ฟังก์ชั่นแบ่งออกเป็น 2 ประเภท

1. ฟังก์ชั่นที่ผู้เขียนโปรแกรมสร้างขึ้นเอง (User Defined Functions)
2. ฟังก์ชั่นที่สร้างจากผู้อื่นถูกเก็บไว้ในไลบรารี (Library) มีทั้งนำมาใช้ได้ฟรี (Open Source) และมีค่าใช้จ่าย (Commercial)

## **การสร้างฟังก์ชั่นขึ้นมาใช้งาน (Creating Function)**

การสร้างฟังก์ชั่นสามารถแบ่งวิธีการสร้างและการทำงานของฟังก์ชั่นออกได้ 4 รูปแบบ คือ

1. การสร้างฟังก์ชั่นที่ไม่มีการส่งค่าและรับค่า

 โปรแกรมแสดงข้อความ Hello โดยไม่มีการส่งค่าและรับค่า

def show\_greeting1():

print ("Hello")

show\_greeting1()

Hello

2.การสร้างฟังก์ชั่นที่ไม่มีการส่งค่าแต่มีการรับค่า

โปรแกรมแสดงข้อความ Hello โดยมีการส่งค่าชื่อ ไต่ไม่มีการรับค่า

def show\_greeting2(name): #จะมีการรับค่ามาใน ()

print ("Hello", name)

show\_greeting2("Doraemon")

Hello Doraemon

3.การสร้างฟังก์ชั่นที่มีการส่งค่าแต่ไม่มีการรับค่า  
โปรแกรมแสดงข้อความ ที่ส่งกลับมาจากฟังก์ชั่นด้วยคำสั่ง return

def show\_greeting3():

greeting ="Hello, how are you?"

return greeting

print (show\_greeting3())

Hello,How are you ?

4.การสร้างฟังก์ชั่นที่มีการส่งค่ากลับและมีการรับค่า  
โปรแกรมที่มีการส่งอาร์กิวเมนต์ให้กับพารามิเตอร์ และมีการส่งค่ากลับมายังโปรแกรมที่เรียกใช้งาน ด้วยคำสั่ง return

def show\_greeting4(name):

greeting\_name = "Hello, " + name + " How are you?"

return greeting\_name

print (show\_greeting4("Doraemon."))

Hello , Doraemon , How are you ?

## **ข้อควรจำ**

## **อาร์กิวเมนต์ (Argument)**

คือ ค่าตัวแปร หรือ ค่าคงที่ ที่ใช้สำหรับส่งค่าไปยังฟังก์ชั่นที่เรียกใช้งาน เพื่อนำไปประมวลผลต่อ ค่าอาร์กิวเมนต์จะถูกแนบส่งไปพร้อมกับชื่อฟังก์ชั่นอยู่ในเครื่องหมายวงเล็บ หากมีอาร์กิวเมนต์หลายตัวจะคั่นด้วยเครื่องหมาย comma (,)

**พารามิเตอร์ (Parameter)**

คือ ตัวแปรที่ประกาศไว้รอรับค่าอาร์กิวเมนต์ อยู่ในเครื่องหมายวงเล็บหลังชื่อฟังก์ชั่น หากมีพารามิเตอร์หลายตัวจะคั่นด้วยเครื่องหมาย comma (,)

### การส่งค่าอาร์กิวเมนต์มี 2 วิธี คือ

1.การส่งค่าข้อมูล (Call by Value)

ค่าอาร์กิวเมนต์จะถูกทำการ copy แล้วส่งให้กับพารามิเตอร์ของฟังก์ชันที่เราเรียกใช้งาน จำนวนพารามิเตอร์ต้งเท่ากับจำนวนอาร์กิวเมนต์ที่ส่งมาให้ การเปลี่ยนแปลงค่าพารามิเตอร์ภายในฟังก์ชัน จะไม่มีผลกระทบค่าอาร์กิวเมนต์เมื่อมีการส่งค่ากลับ การส่งค่าอาร์กิวเมนต์แบบนี้ทำให้สิ้นเปลืองเนื้อที่ในหน่วยความจำ

2.การส่งค่าแบบอ้างอิง (Call by Reference)

ใช้กับข้อมูลชนิดออปเจ็ค โดยส่งค่าตำแหน่งอาร์กิวเมนต์ในหน่วยความจำ ไปให้กับพารามิเตอร์ในฟังก์ชันที่เรียกใช้งาน เมื่อมีการเปลี่ยนแปลงข้อมูลของค่าพารามิเตอร์จะทำให้อาร์กิวเมนต์ที่ส่งไปเปลี่ยนตามไปด้วย การส่งอาร์กิวเมนต์แบบนี้มีข้อดีคือ ประหยัดเนื้อที่ในหน่วยความจำ เพราะค่าอาร์กิวเมนต์ไม่ต้องมีการ copy อีกชุดหนึ่งเหมือนกับวิธีการแบบแรก และลดเวลาในการประมวลผลข้อมูล

การส่งค่าแบบ Call by Value

r = int(input("ป้อนค่ารัศมี : "))

h = int(input("ป้อนค่าความสูง : "))

def cylinder():

area = 3.14 \* (r \* r) \* h

return area

print ("ปริมาตรทรงกระบอก = ", cylinder(), "ตารางเมตร")

### รูปแบบการส่งค่าอาร์กิวเมนต์ให้กับค่าพารามิเตอร์

### การส่งค่าอาร์กิวเมนต์แบบ Required arguments

เป็นการส่งอาร์กิวเมนต์ให้พารามิเตอร์ ตามชนิดข้อมูล และลำดับ โดยต้องมีจำนวนเท่ากันด้วย ดังตัวอย่าง

def req\_arg(numlst, str\_, num):

print ("แสดงค่าข้อมูลในลิสต์ =", numlst)

print ("แสดงค่าข้อมูลในตัวแปร num =", num)

print ("แสดค่าข้อมูลในตัวแปร str\_ =", str\_)

lst = [1,2,3]; msg = "Python"

req\_arg(lst, msg, 50)

แสดงค่าข้อมูลในลิสต์ = [1, 2, 3]

แสดงค่าข้อมูลในตัวแปร num = 50

แสดค่าข้อมูลในตัวแปร str\_ = Python

#### **การส่งค่าอาร์กิวเมนต์แบบ Keyword arguments**

เป็นการส่งอาร์กิวเมนต์ให้พารามิเตอร์ได้โดยไม่ต้องเรียงลำดับ แต่ต้องมีชื่อเดียวกันและต้องมีจำนวนเท่ากัน ดังตัวอย่าง

def key\_arg(lst, msg, num):

print("แสดงค่าข้อมูลในลิสต์ =", lst)

print("แสดงค่าข้อมูลในตัวแปร num =", num)

print("แสดงค่าข้อมูลในตัวแปร msg =", msg)

key\_arg(lst=[1, 2, 3], num=10, msg="Python is easy")

แสดงค่าข้อมูลในลิสต์ = [1, 2, 3]

แสดงค่าข้อมูลในตัวแปร num = 10

แสดงค่าข้อมูลในตัวแปร msg = Python is easy

#### **การส่งค่าอาร์กิวเมนต์แบบ Default arguments**

มีค่าพารามิเตอร์บางตัวถูกกำหนดค่าไว้ล่วงหน้าซึ่งเมื่อเรียกใช้ก็จะส่งค่าอาร์กิวเมนต์เฉพาะจำนวนที่เหลือ ให้พารามิเตอร์เท่านั้น ดังตัวอย่าง

def def\_arg(r, pi=3.14):

result = 2\* pi \* r

print("เส้นรอบวงกลม =", result)

def\_arg(7)

เส้นรอบวงกลม = 43.96

#### **การส่งค่าอาร์กิวเมนต์แบบ Variable-length arguments**

เป็นการสร้างพารามิเตอร์ไว้รับค่าอาร์กิวเมนต์แบบไม่จำกัดจำนวนโดยชื่อพารามิเตอร์ต้องมี เครื่องหมาย (\*)นำหน้า ดังตัวอย่าง

def varleng\_arg(\*num):

print("จำนวนข้อมูลในตัวแปร num :")

for var in num:

print(var, end=" ")

print(" ")

result = num[2] \* 2

print("ผลคูณพารามิเตอร์ num[2] \* 2 =", result)

จำนวนข้อมูลในตัวแปร num :

10 20 30 40

ผลคูณพารามิเตอร์ num[2] \* 2 = 60

## **Lambda Expressions**

Lambda Expressions คือ anonymous function ที่เป็นฟังก์ชันที่มีการทำงานขนาดเล็กอยู่ภายในที่สามารถมีได้เพียง Expression เดียวเท่านั้น เราสามารถสร้างโดยใช้คำสั่ง lambda เราสามารถใช้ Lambda Expressions สร้างออบเจ็คของฟังก์ชันได้ และค่า return จะเป็นค่าที่ได้จากผลลัพธ์ของ Expression ของฟังก์ชัน

f = lambda x: x + 1

print(f(2))

print(f(8))

g = lambda a, b: (a + b) / 2

print(g(3, 5))

print(g(10, 33))

def make\_incrementor(n):

return lambda x: x + n

f = make\_incrementor(13)

print(f(0))

print(f(1))

print(f(5))

ในตัวอย่าง เราได้สร้าง Lambda Expressions เป็นจำนวนสามฟังก์ชัน ฟังก์ชันแรกเป็นฟังก์ชันสำหรับเพิ่มตัวเลขขึ้น 1 และฟังก์ชันที่สองเป็นฟังก์ชันสำหรับหาค่าเฉลี่ยของตัวเลขสองจำนวน คุณจะสังเกตุได้ว่าฟังก์ชันแรกนั้นมี 1 อาร์กิวเมนต์และฟังก์ชันที่สองนั้นมี 2 อาร์กิวเมนต์ และฟังก์ชันสุดท้ายนั้นเป็นการ return ฟังก์ชันกลับภายในฟังก์ชันอีกที และเป็นฟังก์ชันสำหรับเพิ่มตัวเลขขึ้นจำนวน n จากอาร์กิวเมนต์ที่ใส่เข้าไป

3

9

4.0

21.5

13

14

18

นี่เป็นผลลัพธ์การทำงานของโปรแกรม

นอกจากนี้ Lambda Expressions ยังมีประโยชน์เพื่อใช้งานกับ built-in function เช่น ฟังก์ชัน filter() และฟังก์ชัน map() ในภาษา Python มันใช้เป็นอาร์กิวเมนต์ส่งเข้าไปในฟังก์ชัน เพื่อสร้าง Expression ให้กับฟังก์ชัน มาดูตัวอย่างการใช้งาน

numbers = [2, 15, 5, 7, 10, 3, 28, 30]

print(list(filter(lambda x: x % 5 == 0, numbers)))

print(list(map(lambda x: x \* 2, numbers)))

การสร้างฟังก์ชัน lambda ที่ไม่มีการส่งค่าอาร์กิวเมนต์

area = lambda: 3.14 \* 5 \* 5 \* 10

print ("ปริมาตรทรงกระบอก = ", area())

ปริมาตรทรงกระบอก = 785.0

การสร้างตัวแปรโกลบอลและการเรียกใช้งาน โดยฟังก์ชัน cylinder

r = int(input("ป้อนค่ารัศมี : "))

h = int(input("ป้อนค่าความสูง : "))

def cylinder():

area = 3.14 \* (r \* r) \* h

return area

print ("ปริมาตรทรงกระบอก = ", cylinder(), "ตารางเมตร")

ป้อนค่ารัศมี : 3

ป้อนค่าความสูง : 10

ปริมาตรทรงกระบอก = 282.6 ตารางเมตร

การสร้างตัวแปรโลคอลขึ้นมาใช้งานภายในฟังก์ชัน และการเรียกใช้งานตัวแปรข้ามฟังก์ชัน

def circle\_area():

global r; r = 10

global pi; pi = 3.14

result = pi \* (r \* r)

return result

def circumference():

result = 2 \* pi \* r

return result

print ("พื้นที่วงกลม = ", circle\_area(), "ตารางเมตร")

print ("เส้นรอบวงวงกลม = %.3f" %circumference(), "เมตร")

พื้นที่วงกลม = 314.0 ตารางเมตร

เส้นรอบวงวงกลม = 62.800 เมตร

การสร้างตัวแปรโลคอลและโกลบอลที่มีชื่อเหมือนกัน

cash = int(input("ป้อนจำนวนเงิน : "))

vat = 0.07

total = cash - (cash \* vat)

def cash(cash):

total = cash - (cash \* vat)

return total

print ("ชำระเป็นจำนวนเงิน = ", cash(5000), "บาท")

print ("ชำระเป็นจำนวนเงิน = ", total, "บาท")

ป้อนจำนวนเงิน : 520

ชำระเป็นจำนวนเงิน = 4650.0 บาท

ชำระเป็นจำนวนเงิน = 483.6 บาท

#### **สรุป**

ฟังก์ชันเป็นส่วนหนึ่งที่มีความสำคัญในการพัฒนาโปรแกรม ซึ่งในเนื้อหาตอนนี้ได้แนะนำวิธีการสร้างฟังก์ชันขึ้นมาใช้งานแบบต่างๆ เช่น การสร้างฟังก์ชั่นที่ไม่มีการส่งค่า การสร้างฟังก์ชั่นที่ไม่มีการคืนค่า การสร้างฟังก์ชั่นที่มีการรรับค่าแต่ไม่มีการส่งค่ากลับ เป็นต้น นอกจากนี้ยังได้รู้จักกับวิธีการส่งค่าอาร์กิวเมนต์ให้กับค่าพารามิเตอร์ และยังได้รู้จักวิธีการสร้างค่าพารามิเตอร์ไว้รอรับค่าอาร์กิวเมนต์ รวมไปถึงสร้างฟังก์ชันที่ไม่ต้องกำหนดชื่อ ด้วย lambda ทำให้ลดเวลาในการเขียนคำสั่งโปรแกรม ซึ่งที่กล่าวมาทั้งหมดนี้ ไม่ต้องใช้คำสั่ง import เลย