INTRODUCTION TO PYTHON

Prof. Chia-Yu Lin

National Central University

2022 Fall

Python

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- What can Python do?
  - Python can be used on a server to create web applications.
  - Python can be used alongside software to create workflows.
  - Python can connect to database systems. It can also read and modify files.
  - Python can be used to handle big data and perform complex mathematics.
  - Python can be used for rapid prototyping, or for production-ready software development.

Python

- Why Python?
  - Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
  - Python has a simple syntax similar to the English language.
  - Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
  - Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
  - Python can be treated in a procedural way, an object-oriented way or a functional way.

Python 3

- The most recent major version of Python is Python 3, which we shall be using in this tutorial.
  - However, Python 2, although not being updated with anything other than security updates, is still quite popular.

- https://www.python.org/downloads/

How to learn

- 1. https://www.python.org/doc/
- 2. Google “python tutorial”

When you have problem

- Google
- Stackoverflow
- Ask other people to google for you(?)

Whitespace (空白鍵)

- In Python, whitespace is meaningful, especially “tab” or “newline”
- Using “newline” to be the end of a line of text instead of semicolon in C++ or Java.)
- In Python, we don’t use “{}” to represent a block.
- Use consistent indentation instead. That is, you have to use tab or space in the whole program.
- A colon(冒號) usually appears at the beginning of a new block, such as function and class definitions.

```
for train, test in kfold.split(dataset):
    X_train= dataset.iloc[train]
    Y_train = label.iloc[train]
```

Comments(註解)

- Using “#” as the start of a comment
- “ “ “ “ “ “ is comment for many lines.
- Using comments to explain a new defined function.
- The development environment, debugger, and other tools use it:
  it’s good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
   function does blah blah blah."""
   # The code would go here...
```

Understanding Basic Concept

Output

```
print('Hi, my name is', 'Simon')
```

```
print('Now processing',file_name,'...')
```

Variable(變數)

- In Python, you don ‘t need to declare variable.
- Python will define the type of variable according to the initial value.
  - int
  - Float
  - str

```
iv = 10
fv = 12.3
cv = 3 + 5j
sv = 'hello python'
bv = True
nv = None

print(iv, fv, cv, sv, bv)
print(type(iv))
print(type(fv))
print(type(cv))
print(type(sv))
print(type(bv))
print(nv)
print(isinstance(sv, str))
```

User Input

- “input” will return the external message.

```
name = input('Hello, what is your name?  ')
print('Hi, ', name)
```

Understanding Assignment

Names and References

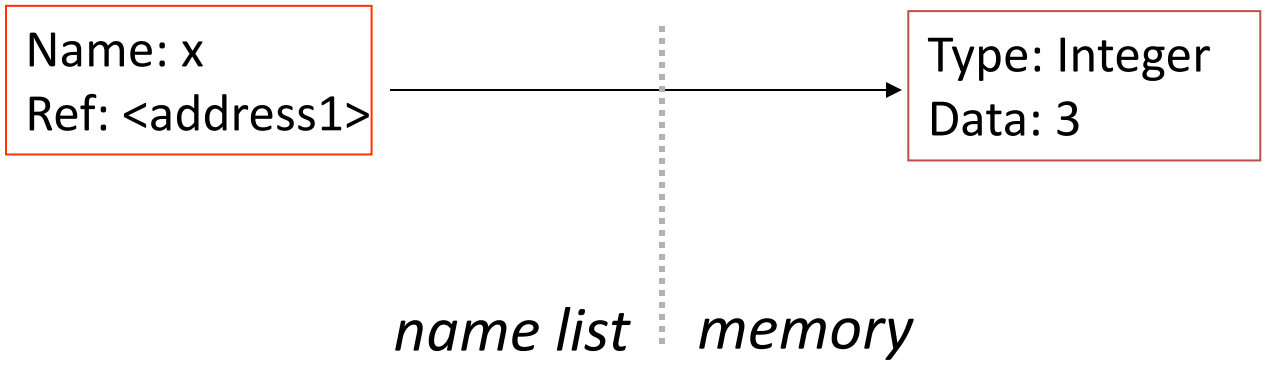
- Python has no pointers like C or C++. Instead, it has "names" and "references". (Works a lot like Lisp or Java.)
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3
```

- Names store "references" which are like pointers to locations in memory that store a constant or some object.
  - Python determines the type of the reference automatically based on what data is assigned to it.
  - It also decides when to delete it via garbage collection after any names for the reference have passed out of scope.

Names and References

- There is a lot going on when we type:
  `x = 3`
- First, an integer 3 is created and stored in memory.
- A name x is created.
- An reference to the memory location storing the 3 is then assigned to the name x.



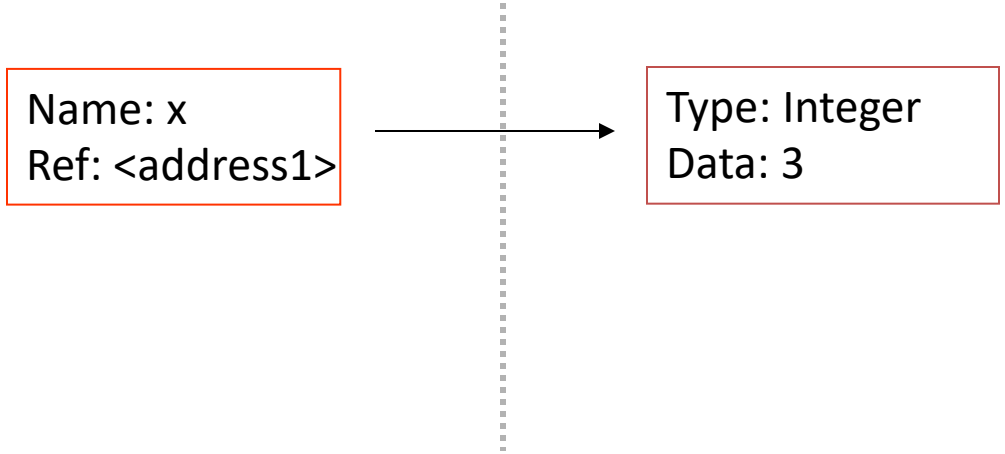
Names and References

- The data 3 we created is of type integer.
- In Python, the basic datatypes integer, float, and string are "immutable."
- This doesn't mean we can't change the value of x…
- For example, we could increment x.

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

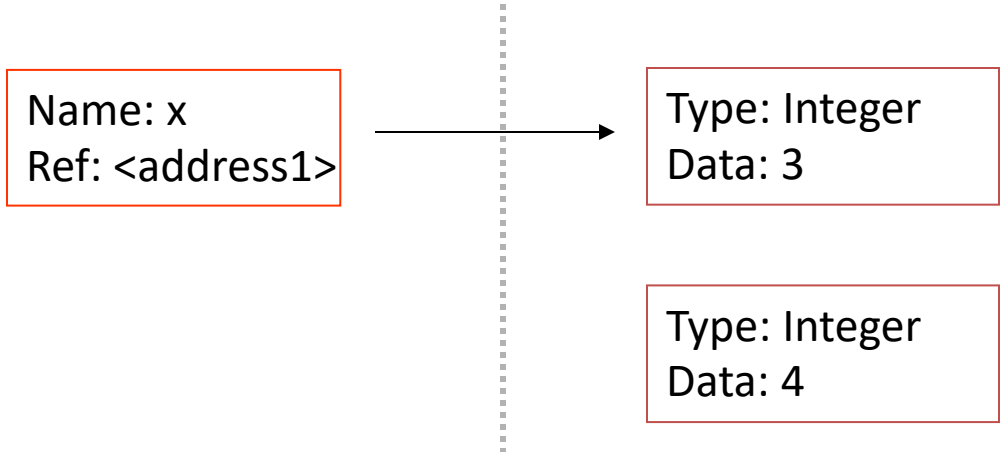
Names and References

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



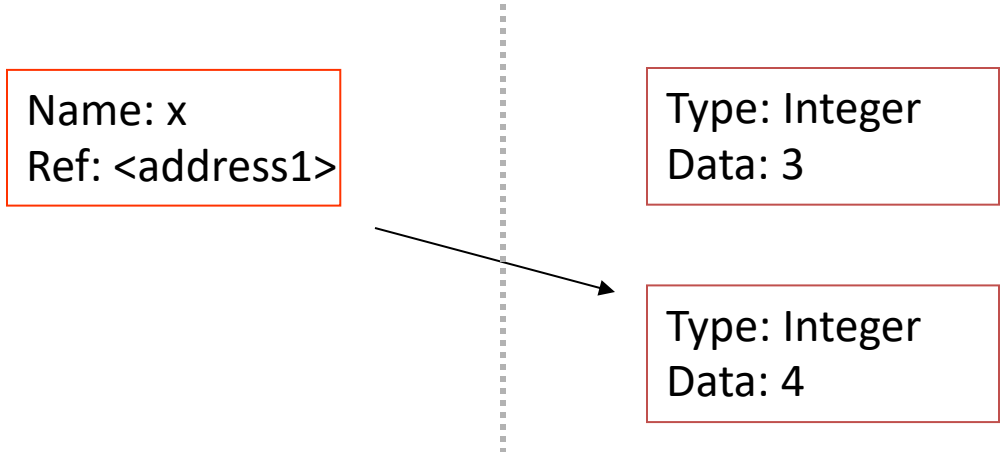
Names and References

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



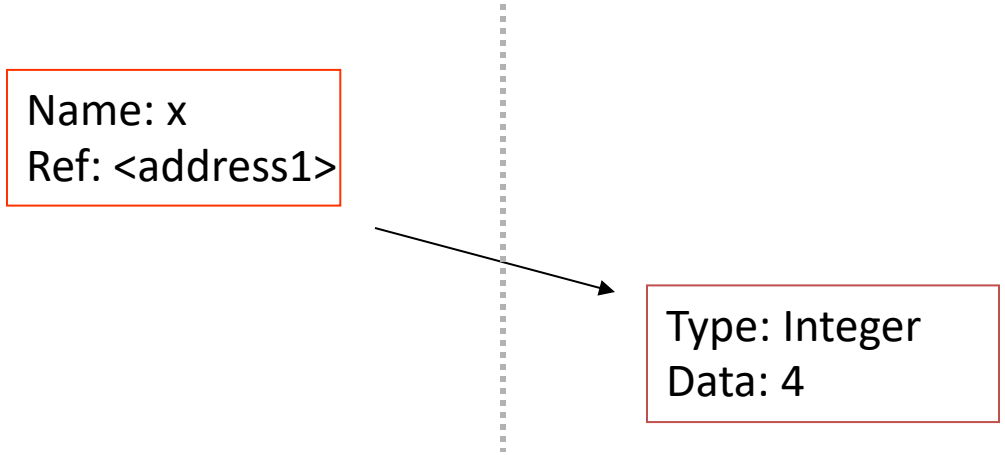
Names and References

- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Names and References

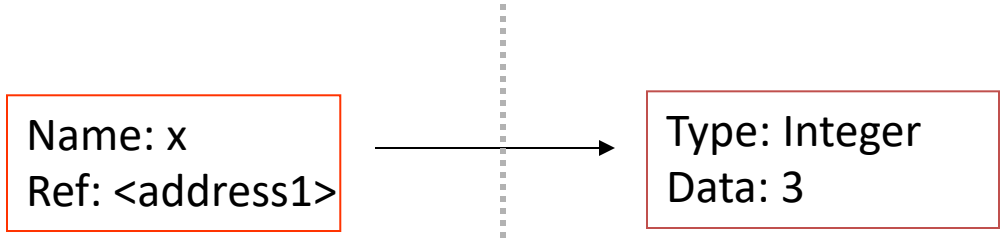
- If we increment x, then what's really happening is:
  - The reference of name x is looked up.
  - The value at that reference is retrieved.
  - The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
  - The name x is changed to point to this new reference.
  - The old data 3 is garbage collected if no name still refers to it.



Assignment

- For simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

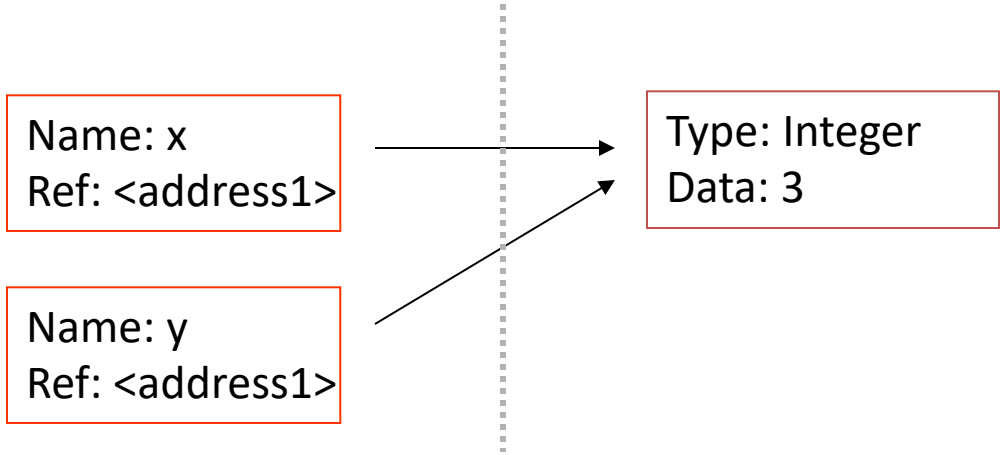
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment

- For simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

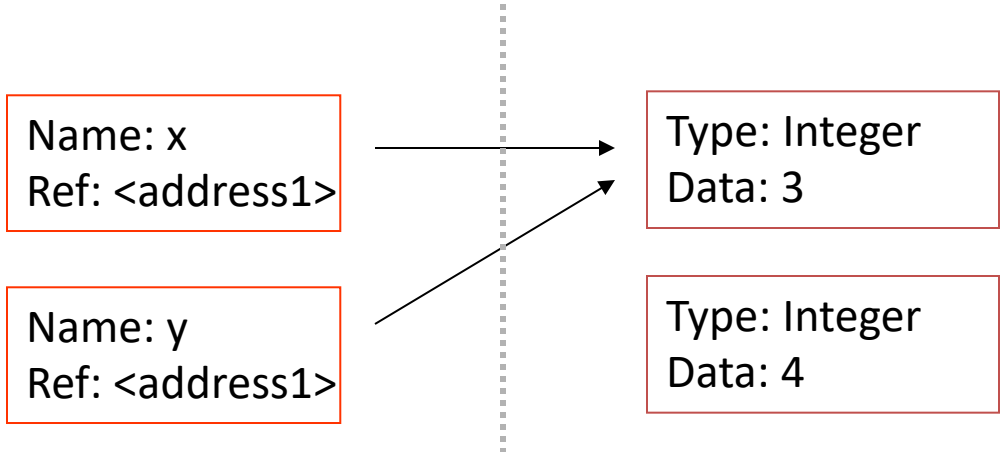
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

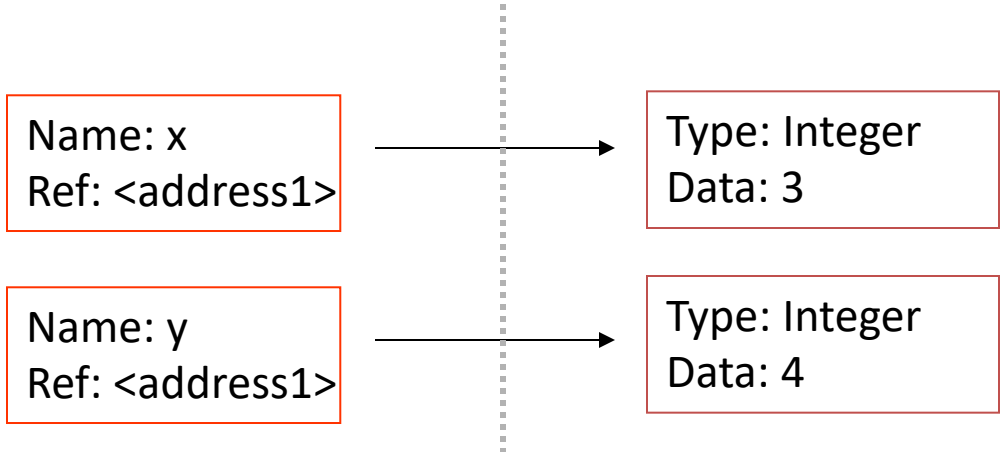
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment

- For other data types (lists, dictionaries, user-defined types), assignment works differently.
  - These datatypes are **"mutable."**
  - When we change these data, we do it *in place.*
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed!
  - We'll talk more about "mutability" later.

immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

x = some mutable object
y = x
make a change to y
look at x
x will be changed as well

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

```
and, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while
```

Accessing Non-Existent Name

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined

>>> y = 3
>>> y
3
```

Multiple Assignment

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Understanding Operator

Mathematical Operator

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Comparison Operator

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Boolean

Operator	Operation
a or b	If a or b is True, return True.
a and b	If a and b are True, return True.
not A	If A is True, return False. Otherwise, return True.

Understanding Container

Range

- Store the variables in specific range.
- The content cannot be modified once range is created.

- Range(stop)
  - stop：停止點
- Range(start, stop)
  - start：起始點
  - stop：停止點
- Range(start, stop, step)
  - start：起始點
  - stop：停止點
  - step：間隔

```
r1 = range(10)
r2 = range(5, 50, 5)

print(type(r1))
print(r1)
print(r2)
```

Store 0~9

- If there is no start, the default start is “0.”
- If there is no step, the default step is “1.”
- Occurring stop, the process will be ended. That is, there is no stop point in the range.

Tuple

- Store a set of data.
- Data in tuple can be different types.
- The content cannot be modified once tuple is created.

```
t1 = 10, 20
# it can hold different types of data
t2 = 10, 'hello world'

print(type(t1))
print(t1)
print(t2)
```

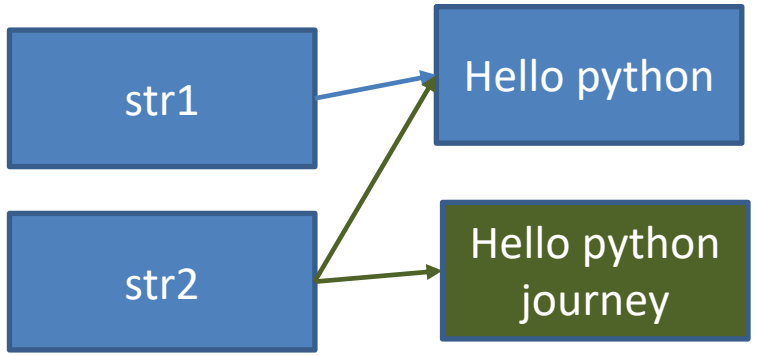
List

- Array of Python
- The type in List can be different.

```
arr1 = [1, 2, 3]
arr2 = [10, 'hello world', 8.7]
arr1[0] = [1, 2, 3]

print(type(arr1))
print(arr1)
print(arr2)
```

String



- The content of string cannot be modified.
- Combine string: "+"
- "is" can be used to see whether two string use the same memory.
- Split & join are two important actions in string type.

```
str1 = 'hello python'
str2 = str1
# str2[0] = 'y'
# a = a + b could be written as a += b
str2 += ' journey'
print(str2 is str1)

print(str1)
result = str2.split(' ')
print(result)
result_back = '***'.join(result)
print(result_back)
```

['hello', 'python', 'journey']

Hello***python***journey

Similar Syntax

- Tuples and lists are sequential containers that share much of the same syntax and functionality.
  - For conciseness, they will be introduced together.
  - The operations shown in this section can be applied to both tuples and lists, but most examples will just show the operation performed on one or the other.

- While strings aren’ t exactly a container data type, they also happen to share a lot of their syntax with lists and tuples; so, the operations you see in this section can apply to them as well.

Tuples, Lists, and Strings

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (“, ‘).

```
>>> st = "Hello World"
>>> st = 'Hello World'
```

Tuples, Lists, and Strings

- We can access individual members of a tuple, list, or string using square bracket "array" notation.

```
 tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
'abc'

li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
34

st = "Hello World"
>>> st[1]    # Second character in string.
'e'
```

Looking up an Item

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Container

You can make a copy of the whole tuple using `[:]`.

```
>>> t[:]
   (23, 'abc', 4.56, (2,3), 'def')
```

So, there's a difference between these two lines:

```
>>> list2 = list1    # 2 names refer to 1 ref
                     # Changing one affects both

>>> list2 = list1[:]    # Two copies, two refs
                        # They're independent
```

Get Data from Sequence

- Sequence Types — list, tuple, range
- seq[start:stop:step]
- Step default is 1.
- -1 represents the last variable.
- Start default is 0.
- If you do not set ‘stop’, you can get all variables after start.

```
str1 = 'hello world'
arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# mind the stop
arr2 = arr1[0:5]
# -1 represent the last element
arr3 = arr1[0:-1:2]
# you can ignore the args...
arr4 = arr1[:]

print(arr2)
print(arr3)
print(arr4)
print(arr4 is arr1)
print(str1[5:])
# print(arr1[:-1])
```

Sequence Type Operation

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

https://docs.python.org/3/library/stdtypes.html

Sequence Type Operation

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* in place

https://docs.python.org/3/library/stdtypes.html

What’ s the difference between tuples and lists?

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item
 assignment
```

- You're not allowed to change a tuple *in place* in memory; so, you can't just change one element of it.
- But it's always OK to make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (1, 2, 3, 4, 5)
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place*. So, it's ok to change just one element of a list.
- Name li still points to the same memory reference when we're done.

Operations on Lists Only

```
>>> li = [1, 2, 3, 4, 5]

>>> li.append('a')
>>> li
[1, 2, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 2, 'i', 3, 4, 5, 'a']
```

Operations on Lists Only

The 'extend' operation is similar to concatenation with the + operator. But while the + creates a fresh list (with a new memory reference) containing copies of the members from the two inputs, the extend operates on list **li** in place.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Extend takes a list as an argument. Append takes a singleton.

```
>>> li.append([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [9, 8, 7]]
```

Operations on Lists Only

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')      # index of first occurrence
1

>>> li.count('b')      # number of occurrences
2

>>> li.remove('b')     # remove first occurrence
>>> li
 ['a', 'c', 'b']
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
[8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
[2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- We can always convert between tuples and lists using the list() and tuple() functions.

```
li = list(tu)
tu = tuple(li)
```

Understanding Ifelse

If….else

- Conditions don’ t need to put in ().
- Add “:” after every condition.
- Use “**indentation**” to represent the action if condition is achieved.
- else if in Python is “**elif**”

```
grade = 90

# there's no ()
if grade >= 90:
    print('Excellent!')
elif grade >= 60:
    print('Good enough!')
else:
    print('Loser!')
```

Understanding Loop

for

- For in Python is designed for getting elements from container.

```
arr1 = [2, 4, 6, 8, 10]
str1 = 'hello python'

for i in range(10):
    print(i)
print('***\n')

for i in range(len(arr1)):
    print(arr1[i])
print('***\n')

for i in arr1:
    print(i)
print('***\n')

for i in str1:
    print(i)
print('***\n')

# for i in arr1:
#     i += 1
# print(arr1)
```

If the lens of arr1 is 5, generate 0~4 in range.

C++ vs. Python

C++

```cpp
int getMax(size_t size, int const* array){
    // Find the maximum element in array
    /*
      It is just an example.
      Python has nice build-in function
      called "max()".
    */
    int max = -1 * INT_MAX;
    for (size_t i=0; i<size; ++i) {
        if (array[i] > max)
            max = array[i];
    }
    return max;
}
```

Python

```python
def getMax(array):
    # Find the maximum element in array
    """
      It is just an example.
      Python has nice build-in function
      called "max()".
    """
    max = -1* sys.maxint -1
    for element in array:
        if element > max:
            max = element
    return max
```

Practice

- Leetcode https://leetcode.com/
- **Problem 1313: Decompress Run-Length Encoded List**
- **Problem 1431: Kids With the Greatest Number of Candies**
- **Problem 1480: Running Sum of 1d Array**
- **Problem 1528: Shuffle String**
- **Problem 1672: Richest Customer Wealth**
- **Problem 1512: Number of Good Pairs**

HW

- Capture your pass result of six problems on Leetcode in the word document.
- And use “zip” to hand in your six programs.