CS112 Introduction to Python Programming

Mid-term Review

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

- Mid-term Recapitulation
- Assignment Review
- Quiz Review

Basics

- A **statement** is an instruction that the Python interpreter can execute.
- **Expression** is an arrangement of values and operators which are evaluated to make a new value.
- **Variable** is a named placeholder to hold any data.
- In Python, there is **no need to declare a variable explicitly** by specifying whether the variable is an integer or a float or any other type.
- Variables are labels that you can assign to values. Every variable is connected to a value, which is the information associated with that variable.
- You can change the value of a variable in your program at any time, and Python will always keep track of its current value.
- "=" is the assignment operator, it assigns the value of the number on the right to the variable name on the left

```
>>> message = "Hello Python world!"
>>> print(message)
```

Variable Rules

SUSTech Southern University of Science and Technology

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable `message_1` but not `1_message`.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, `greeting_message` works, but `greeting message` will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print.
- Variable names should be short but descriptive. For example, `name` is better than `n`, `student_name` is better than `s_n`, and `name_length` is better than `length_of_persons_name`.
- Be careful when using the lowercase letter `l` and the uppercase letter `O` because they could be confused with the numbers `1` and `0`.

Be sure to keep the variable rules in mind, breaking some of these rules will cause errors.

Keywords

- Keywords are a list of reserved words that have predefined meaning.
- Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Operators

SUSTech Southern University of Science and Technology

- Arithmetic operators

addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
floor division	`//`
remainder	`%`
exponentiation	`**`

- Comparison operators

equal to	`==`
not equal to	`!=`
greater than	`>`
lesser than	`<`
greater than or equal to	`>=`
lesser than or equal to	`<=`
`is` operator	

- Logical operators

Logical AND	`and`
Logical OR	`or`
Logical NOT	`not`

P	Q	P and Q	P or Q	Not
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Data Types

- **Integer**s: can be positive or negative; a number is automatically treated as an integer if it is written without a decimal
- **Float**ing numbers: numbers with a decimal point
- **Complex** numbers: numbers with a real and imaginary part
- **Boolean**: values can be only `True` or `False`
- **String**s: A sequence of one or more characters; includes letters, numbers, and other types of characters such as spaces
- **None:** `None` is used to represent the absence of a value
- Data types can be converted via type conversions with corresponding type functions:
    - `int()`
    - `float()`
    - `str()`
    - `chr()`
    - `complex()`

Numbers

- Python treats numbers in several different ways, depending on how they're being used.
- You can add (+), subtract (-), multiply (*), and divide (/) `integers` in Python.
- Python supports the order of operations, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations.
- Python calls any number with a decimal point a `float`.
- Python defaults to a `float` in any operation that uses a `float`, even if the output is a whole number.
- When you're writing long numbers, you can group digits using underscores to make large numbers more readable. Python ignores _ in numbers and prints only the digits.

```
>>> universe_age = 14_000_000_000
>>> print(universe_age)
```

```
14000000000
```

- A constant is like a variable whose value stays the same throughout the life of a program. Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed.

```
>>> MAX_CONNECTIONS = 5000
```

Strings

- A string is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings.
- This flexibility allows you to use quotes and apostrophes within your strings.

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
```

- Strings can be concatenated using the "+" operator, and can be repeated using the "*" operator.

immutable

- Strings can be joined with the `.join()` method.
- Strings can be split with the `.split()` method into list of strings based on the specified separator; if the separator is not specified then whitespace is considered as the delimiter.
- `.format()` method can be used to insert the value of a variable, expression or an object into another string.
- A string prefix with a "`f`" is using `f-string` format:
  - **`f"string_statements {variable_name [: {width}.{precision}]}"`**
    - Using *variable_name* along with either *width* or *precision* values should be separated by a colon
    - Precision refers to the total number of digits that will be displayed in a number
    - By default, strings are left-justified and numbers are right-justified

Strings

- A raw string is created by prefixing the character `r` to the string
- A raw string ignores all types of formatting within a string including the escape characters
- `in` and `not in`: check for the presence of a string in another string
- Strings are compared based on the ASCII value of the characters using `>` , `<` , `<=` , `>=` , `==` , `!=`
- As strings are immutable, they cannot be modified.
- The characters in a string cannot be changed once a string value is assigned to string variable. However, you can assign different string values to the same string variable.
- You can use `len()` to check number of characters in a string, use `min()` and `max()` to get the character having the lowest or highest ASCII value.
- Each of the string's character corresponds to an index number starting from 0; square brackets are used to perform indexing in a string.
- String slicing returns a sequence of characters beginning at **`start`** and extending up to but not including **`end`**; the index numbers are separated by a **`colon`**.
- A third argument called **`step`** can be specified along with the **`start`** and **`end`** index numbers to specify steps in slicing; the default value of **`step`** is 1.

Strings

- Escape sequences are a combination of a backslash (\) followed by either a letter or a combination of letters and digits.
- A method is an action that Python can perform on a piece of data.
- The dot (`.`) after `name` in `name.title()` tells Python to make the `title()` method act on the variable `name`. Every method is followed by a set of parentheses, because methods often need additional information provided inside the parentheses.
- The `.title()` method changes each word to title case, where each word begins with a capital letter. The `.lower()` or `.upper()` method changes each word to lower or upper case.
- Python can trim extra whitespace on the right (`.rstrip()`), left (`.lstrip()`), or both sides (`.strip()`) of a string.

```
>>> name = "ada lovelace"
>>> print(name.title())
>>> print(name.upper())
```

```
Ada Lovelace
ADA LOVELACE
```

```
>>> favorite_language = 'python '
>>> print(favorite_language)
>>> print(favorite_language.rstrip())
```

```
'python '
'python'
```

Lists

- A `list` is a collection of items in a particular order.
- In Python, square brackets (`[]`) indicate a `list`, and individual elements in the list are separated by commas.
- If you ask Python to print a list, Python returns its representation of the list, including the square brackets.

```
>>> bicycles = ['trek', 'cannondale', 'redline']
>>> print(bicycles)
>>> print(bicycles[0])
```

```
['trek', 'cannondale', 'redline’]
'trek'
```

- Lists are ordered collections, so you can access any element in a list by telling Python the position, or index, of the item desired, enclosed in square brackets.
- Python considers the first item in a list to be at position 0, not position 1.
- By asking for the item at index -1, Python always returns the last item in the list. This convention extends to other negative index values as well (-2, -3, …).
- Lists are dynamic/mutable in nature, meaning you’ll build a list and then add and remove elements from it as your program runs its course.

```
>>> motorcycles = ['honda', 'yamaha', 'suzuki’]
>>> print(motorcycles)
>>> motorcycles[0] = 'ducati'
```

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

Lists

- `.append()` method will add a new element to the end of a list.
- A new element can be added at any position by using the `.insert()` method. It takes an index and the value of the new item to be inserted, shifting items after to the right.

```
>>> motorcycles = ['honda', 'yamaha', 'suzuki']
>>> motorcycles.insert(0, 'ducati’)
>>> print(motorcycles)
```

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

- If you know the position of the item you want to remove from a list, you can use the `del` statement.
- The `.pop()` method removes the last item in a list and returns the item.
- The term pop comes from thinking of a list as a stack of items and popping one item off the top of the stack. The top of a stack corresponds to the end of a list.
- You can use `.pop()` to remove an item from any position in a list by including the ~~index~~ of the item you want to remove in parentheses.
- `.remove()` method removes an item by value in a list. It deletes only the first occurrence of the value you specify.

"remove + while-loop"

- `.count()` counts the number of times the item has occurred in the list and returns it
- `.index()` returns the index for the given item from the start of the list
- `.copy()` creates a new copy of the existing list

Lists

SUSTech Southern University of Science and Technology

- The `.sort()` method changes the order of the list permanently, in alphabetical order by default.
- You can also sort the list in reverse alphabetical order by passing the argument `reverse=True` to the `.sort()` method.
- The `sorted()` function returns you a sorted list in a particular order but doesn't affect the actual order of the list.
- To reverse the original order of a list, you can use the `.reverse()` method.
- You can quickly find the length of a list by using the `len()` function.
- Lists can be concatenated using the `+` sign, and the `*` operator is used to create a repeated sequence of list.
- Lists can be sliced using the `[start:end:stepsize]` syntax, default `stepsize=1`
- To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`).

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> cars.sort()                                        ['audi', 'bmw', 'subaru', 'toyota']
>>> print(cars)

>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> cars.sort(reverse=True)                            ['toyota', 'subaru', 'bmw', 'audi']
>>> print(cars)
```

Lists

- `range()` creates a uniformly spaced sequence of integers. Its general form is `range([start,] ~~stop~~[, step])`:
- The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide, but not include it.
- If you pass a third argument to `range()`, Python uses that value as a step size when generating numbers. It tells Python how many items to skip between items in the specified range. The default step size is 1.
- The `for` statement tells Python to retrieve the value element-wise from the list, and associate it with the temperate variable name.
- Any lines of code after the `for` loop that are not indented are executed once without repetition.
- A list comprehension allows you to generate this same list in just one line of code.
- A list comprehension combines the for loop and the creation of new elements into one line and automatically appends each new element.

Dictionaries

- A dictionary in Python is a collection of **key-value pairs**. Each key is connected to a value, and you can use a key to access the value associated with that key.
- A dictionary is wrapped in braces, { }, with a series of key-value pairs inside the braces. A key-value pair is a set of values associated with each other.
- Any object in Python can be used as a value in a dictionary, but keys have to be immutable objects.
- To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets.

```
>>> alien_0 = {'color': 'green'}
>>> print(alien_0['color'])
```

- Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time.

```
>>> alien_0['x_position'] = 0
>>> alien_0['y_position'] = 25
```

- As of Python 3.7, dictionaries retain the order in which they were defined. When you print a dictionary or loop through its elements, you will see the elements in the same order in which they were added to the dictionary.
- To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key.

Dictionaries

- The `del` statement completely remove a key-value pair in a dictionary. All `del` needs is the name of the dictionary and the key that you want to remove.
- It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.
- The `.get()` method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist. By default, the second argument returns the value `None`.
- You can loop through all of a dictionary's key-value pairs (`.items()`), through its keys (`.keys()`), or through its values (`.values()`) using the `for` statement.
- To check the presence of a key in the dictionary, use `in` or `not in` membership operators.
- You can use the `sorted()` function to get a copy of the keys in order.
- `.fromkeys()` creates a new dictionary from the given sequence of elements with a value provided by the user.
- `.update()` updates the dictionary with the `key:value` ~~pairs~~ from other dictionary object.
- `.pop()` removes the key from the dictionary and returns its value.

Dictionaries

- `.popitem()` removes and returns a pair of `(key, value)` as a tuple from the dictionary.
- `.clear()` removes all the `key:value` pairs from the dictionary.
- The dictionary can also be built using `dictionary_name[key] = value` syntax by adding `key:value` pairs to the dictionary
- Sometimes you want to store multiple dictionaries in a list, or a list of items as a value in a dictionary. This is called **nesting**.
- You should not nest lists and dictionaries too deeply. If there is a significant level of nesting, likely a simpler way to solve the problem exists.

```
favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
    }

for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
    for language in languages:
        print(f"\t{language.title()}")
```

```
Jen's favorite languages are:
    Python
    Ruby
Sarah's favorite languages are:
    C
Phil's favorite languages are:
    Python
    Haskell
Edward's favorite languages are:
    Ruby
    Go
```

Tuples

SUSTech Southern University of Science and Technology

- Tuples are lists that are immutable, i.e., once a tuple is created, you cannot change its values
- A tuple is a finite sequence of ordered, immutable, and heterogeneous items that are of fixed size enclosed in parentheses `()` , while list uses square brackets `[]`.
- Tuples can be created without any values: `empty_tuple = ()`
- Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple ~~with one element~~, you need to include a trailing comma: `my_t = (3,)`
- Individual items of a tuple are addressed in the same way as those of lists, but the elements cannot be changed.
- Although you can’t modify a tuple, you can assign a new value to a variable that represents a tuple by redefining the entire tuple.
- The `+` or `*` operator concatenates or repeats a tuple and returns a new tuple.
- The `in` and `not in` membership operators check for the presence of an item in a tuple.
- Comparison operators like `<`, `<=`, `>`, `>=`, `==` and `!=` can also be used to compare tuples, position by position.

Tuples

SUSTech Southern University of Science and Technology

- The built-in `tuple()` function is used to create a `tuple` from an `iterable` object: i.e.: `tuple("sustech") => ('s', 'u', 's', 't', 'e', 'c', 'h')`
- `len()` function returns the numbers of items in a tuple
- `sum()` function returns the sum of numbers in the tuple
- `sorted()` function returns a sorted copy of the tuple as a list while leaving the original tuple untouched
- The `.count()` method counts the number of times the item has occurred in the tuple and returns it
- The `.index()` method searches for the given item from the start of the tuple and returns its first appearance index
- A tuple can nest other tuples, or any type of items, or lists
- If an item within a tuple is mutable, then you can change it. Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple
- Tuples can be used as `key:value` pairs to build dictionaries, which is achieved by nesting tuples within tuples, wherein each nested tuple item should have two items (the first item becomes the key and the second item as its value)

Tuples

- Tuple creation is a process of tuple packing, and the reverse process of assigning tuple elements to multiple variables is tuple unpacking

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
```

```
>>> x, y, z = t
>>> z
'hello!'
```

- Tuple unpacking requires that there are as many variables on the left side of the equals sign as there are items in the tuple.
- Tuples are very useful when returning more than one value from a function, or passing multiple values through one function parameter
- The `zip()` function returns a sequence of tuples, where the i-th tuple contains the i-th element from each of the iterables. The aggregation of elements stops when the shortest input iterable is exhausted

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6, 7]
>>> zipped = zip(x, y)
>>> zipped
<zip object at 0x0000022AA66C3BC8>
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
```

Sets

- A `set` is an unordered collection in which each item must be unique. Keys of a dictionary are a set of non-duplicated immutable elements.

```
>>> languages = {'python', 'ruby', 'c'}
```

- You can build a set directly using curly braces `{}` and separating the elements with commas.
- The `set()` function can be used to create sets from lists, tuples or dictionaries.
- To create an empty set, you must use `set()` but NOT `{}`, as the latter creates an empty dictionary
- Sets are mutable, but set items are unordered, meaning it cannot be indexed or sliced
- Primary uses of sets include membership testing and eliminating duplicate entries
- Set operations (`-`, `|`, `&`, `^`) can also be done with set methods

```
>>> set1 = {"a", "b", "e", "f", "g"}
>>> set2 = {"a", "e", "c", "d"}
>>> set2.difference(set1)
{'c', 'd'}
>>> set2.intersection(set1)
{'a', 'e'}
>>> set2.union(set1)
{'a', 'c', 'd', 'b', 'e', 'f', 'g'}
>>> set2.symmetric_difference(set1)
{'c', 'g', 'd', 'b', 'f'}
```

```
>>> set1 = {"a", "b", "e", "f", "g"}
>>> set2 = {"a", "e", "c", "d"}
>>> set2.add("h")
>>> set2
{'h', 'a', 'c', 'd', 'e'}
>>> set2.update(set1)
>>> set2
{'h', 'b', 'f', 'a', 'c', 'd', 'g', 'e'}
>>> set1.discard("a")
>>> set1.clear()
```

Add an element to a set.

Update a set with the union of itself and others.

Remove an element from a set

Remove all elements from a set

NumPy Array

- A NumPy array is similar to a list but all the elements should be of the same type (homogeneous).
- The elements of a NumPy array are usually numbers, but can also be Booleans, strings, or other objects
- When the elements are numbers, they must all be of same type, e.g., they might be all integers or all floating numbers
- Each dimension of an array has a length which is the total number of elements in that direction. The size of an array is the total number of elements contained in an array
- The size of NumPy arrays are fixed; once created it cannot be changed again
- The `array` function can convert a list or tuple to a NumPy array
- NumPy promotes all the numbers to the most general type
- The `np.zeros()` and `np.ones()` function to create arrays where all the elements are either zeros or ones, with one mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array.
- The `np.arange(start, stop, step)` function produces evenly spaced points between `start` (0) and `stop` (not included) with `step` (1) increment.

```
>>> import numpy as np
>>> a = [0, 0, 1, 4]
>>> b = np.array(a)
>>> b
array([0, 0, 1, 4])
```

NumPy Array

- The `np.linspace()` function creates an array of N (num) evenly spaced points between a starting point and an ending point. The form of the function is `np.linspace(start, stop, num)`. If the third argument num is omitted, then `num=50`.
- The `np.logspace` function produces evenly spaced points on a logarithmically spaced scale. The form of the function is `logspace(start, stop, num)`. The start and stop refer to a power of 10, i.e., the array starts at 10^{start} and ends at 10^{stop}.

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
>>> np.set_printoptions(precision=1)
>>> np.logspace(1, 3, 5)
array([  10. ,   31.6,  100. ,  316.2,
1000. ])
```

- Basic array attributes:

```
>>> arr.ndim              # Number of array dimensions
>>> arr.shape             # Tuple of array dimensions
>>> arr.size              # Number of elements in the array
>>> arr_att.dtype         # Data-type of the array's elements
>>> arr_att.itemsize      # Length of one array element in bytes
```

NumPy Array

- One-dimensional arrays can be indexed and sliced the same way as strings and lists
- Multi-dimensional arrays can be indexed and sliced per axis:

```
>>> a = np.array([[1,2,3,4],
    [5,6,7,8], [9,10,11,12]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> a[1, 3]
8
>>> a[1][3]
8
>>> a[:2, 1:3]
array([[2, 3],
       [6, 7]])
```

- Boolean indexing can be used to reassign values of an array that meet some criteria

```
>>> a = np.linspace(-1., 5, 5)
>>> a
array([-1. ,  0.5,  2. ,  3.5,  5. ])
>>> a[a>1.]
array([2. , 3.5, 5. ])
>>> a>1.
array([False, False,  True,  True,  True])
>>> a[a>1.] = 1.
>>> a
array([-1. ,  0.5,  1. ,  1. ,  1. ])
```

- Mathematical operations with arrays are vectorized because the entire array, or "vector," is processed as a unit. `np.add(a, b)  np.subtract(a, b)  np.multiply(a, b)  np.divide(a, b)`
- Axes are defined for arrays with more than one dimension.
- A 2-dimensional array has two corresponding axes:
  - the first running vertically downwards across rows (axis=0),
  - the second running horizontally across columns (axis=1).

```
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

0	1	1
0	5	5
0	6	

`if` statement

- **Loops:** Computer programs are useful for performing repetitive tasks. A loop is a statement or a block of statements that is executed repeatedly with minor, but important, variations
- **Conditions:** In the course of doing these repetitive tasks, computers often need to make decisions. Conditional statements allow a computer program to take different actions based on whether some condition, or set of conditions is true or false. In this way, the programmer can control the flow of a program.
- Conditionals and loops control the flow of a program. They are essential to performing virtually any significant computational task.
- The `if` decision control flow statement starts with the `if` keyword and ends with a colon
- The `if` block statements are determined through indentation and the first unindented statement marks the end
- At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a **conditional test**, this expression is a **Boolean expression**
- If a conditional test evaluates to `True`, Python executes the code following the `if` statement.
- Use the optional `else` block to execute statements if the Boolean expression is `False`

`if` statement

- A double equal sign (==), this equality operator returns `True` if the values on the left and right side of the operator match and `False` if they don't match.
- The inequality operator, an exclamation point and an equal sign (`!=`), determines whether two values are not equal.
- You can include various mathematical comparisons in your conditional statements as well, such as less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).
- The keywords `and` and `or` to combine two or more conditional tests.
- To find out whether a particular value is already in a list, use the keyword `in`.
- The `not in` keyword can be used to check if a value does not appear in a list.
- The simplest kind of `if` statement has one test and one action.
- If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

```
if conditional_test:
    do something
```

- The `if-else` statement allows for a two-way decision, the `else` statement allows you to define an action or set of actions that are executed when the conditional test fails.

`if` statement

- The `if-else` structure works well in situations in which you want Python to always execute one of two possible actions.
- The `if-elif-else` is a multi-way decision control statement, it tests more than two possible situations. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes, and skips the rest of the tests.
- There can be zero or more `elif` parts each followed by an indented block, but can be only one `else` block
- Python does not require an `else` block at the end of an `if-elif` chain.
- The `else` block is a catchall statement. It matches any condition that wasn't matched by a specific `if` or `elif` test.
- An `if` statement that contains another `if` statement either in its `if` block or `else` block is called a nested `if` statement
- To check all of the conditions of interest, a series of simple `if` statements with no `elif` or `else` blocks may be used. This technique makes sense when more than one condition could be `True`, and you want to act on every condition that is `True`.
- In summary, if you want only one block of code to run, use an `if-elif-else` chain. If more than one block of code needs to run, use a series of independent `if` statements.

`for` loops

- Python has two kinds of loops
  - a `for` loop, and
  - a `while` loop
- The `for` loop starts with the `for` keyword and ends with a colon
- The for loop assigns items from the sequence to the iteration_variable and then executes the statements one by one until all the items in the sequence are completed

```
for iteration_variable in sequence:
    statement(s)
```

- Whenever the `break` statement is encountered, the execution control immediately jumps to the first instruction following the loop
- To pass control to the next iteration without exiting the loop, use the `continue` statement
- `break` jump out of the loop while `continue` skip the current iteration

`while` loops

- The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or while, a certain condition is true.
- The `while` loop starts with the `while` keyword and ends with a colon
- The Boolean expression is evaluated before the statements in the `while` loop block is executed
- After each iteration of the loop block, the Boolean expression is again checked, and if it is `True`, the loop is iterated again
- This process continues until the Boolean expression evaluates to `False` and at this point the `while` statement exits:

```
while Boolean_Expression:
    statement(s)
```

- Every `while` loop needs a way to stop running so it won't continue to run forever.
- A `for` loop is effective for looping through a list, but you shouldn't modify a list inside a `for` loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a `while` loop.

Functions

- Functions are used when you have a block of statements that needs to be executed multiple times within the program.
- This block of statements are grouped together and is given a name which can be used to invoke it from other parts of the program.
- User-defined functions are reusable code blocks created by users to perform some specific task in the program:

```
def function_name(parameter_1,…, parameter_n):
    statement(s)
```

- In Python, a function definition consists of the **def** keyword, followed by:
  The function name: use letters, numbers, or an underscore, but the name cannot start with a number
  A list of parameters enclosed in "()" and separated by ",". Some functions may not have parameters
  A colon at the end of the function header
  Block of statements that define the body of the function start and they must have the same indentation level
- Arguments are the actual value that is passed into the calling function. An argument is a piece of information that's passed from a function call to a function.
- A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function.

Functions

- You can pass arguments to your functions in a number of ways.
- You can use **positional arguments**, which need to be in the same order the parameters were written; **keyword arguments**, where each argument consists of a variable name and a value; and **lists** and **dictionaries** of values.
- **Positional:** Python match each argument in the function call with a parameter in the function definition based on the order of the arguments provided.
- **Keyword**: A keyword argument is a name-value pair that you pass to a function.
- When writing a function, you can define a default value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value.
- Using default values can simplify your function calls and clarify the ways in which your functions are typically used. You can use default values to make an argument optional.
- Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.
- If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in `__name__` variable to have a string value `"__main__"`:

Functions

- The value the function returns is called a **return value**. The `return` statement terminates the function definition and returns to the calling function with an optional value
- Return values allow you to move much of your program’s grunt work into functions, which can simplify the body of your program.
- A function can return any value you need it to, including more complicated data structures like lists and dictionaries.
- It is possible to define functions without a `return` statement. Functions like this are called void functions, and they return `None`.
- A function can return only a single value, but that value can be ~~a list or tuple~~.
- When returning multiple values, separating them by a comma will by default construct a tuple by Python.
- Python programs have two scopes: global and local.
- Global variable is accessible and modifiable throughout the program. A variable that is defined inside a function definition is a local variable
- The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition

Functions

- A function definition can be nested within another function definition
- The inner function can use the arguments and variables of the outer function
- The inner function definition can be invoked by calling it from within the outer function definition
- Changes to immutable arguments of a function within the function do not affect their values in the calling program
- Lists and arrays are mutable; those elements that are changed inside the function are also changed in the calling function
- The function can modify the list. Any changes made to the list inside the function’s body are permanent.
- You can send a copy of a list to a function like this: `function_name(list_name[:])`, if you want to keep the original list untouched.
- It’s more efficient for a function to work with an existing list to avoid using the time and memory needed to make a separate copy, you should pass the original list to functions unless you have a specific reason to pass a copy.
- Numbers, strings and tuples are immutable; changing them within a function creates new objects with the same name inside of the function, but the old objects remain unchanged.

Functions

- Sometimes you won't know ahead of time how many arguments a function needs to accept. `*args` as parameter in function definition allows you to pass a non-keyworded & variable length tuple argument list to the function definition.
- The asterisk in the parameter name `*args` tells Python to make an empty tuple called `args` and pack whatever values it receives into this tuple.
- Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. `**kwargs` as parameter in function definition allows you to pass keyworded & variable length dictionary argument list to the function definition.
- `*args` must come after all the positional parameters and `**kwargs` must come right at the end.
- You can go a step further by storing your functions in a separate file called a module and then importing that module into your main program. An import statement tells Python to make the code in a module available in the currently running program file.

Functions

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
Making a pizza with the following toppings:
-  Pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein’,
                             location='princeton’,
                              field='physics')
print(user_profile)
```

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

Input

- The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable.
- Each time you use the `input()` function, you should include a clear, easy to follow prompt that tells the user exactly what kind of information you're looking for. Any statement that tells the user what to enter should work.
- You can assign your prompt to a variable and pass that variable to the `input()` function.

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "
name = input(prompt)
print(f"\nHello, {name}!")
```

- When you use the `input()` function, Python interprets everything the user enters as a string.
- By using the `int()` or `float()` function, Python converts the input to numeric.

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)
if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

Southern University of Science and Technology

Assignments

Assignment 1-A

Lower Cases

Description

- Given a string, display it in lower cases.
- Example:

Input #1	Output #1
ApPlE	apple

```
# Solve:
string = input()
print(string.lower())
```

```
>>> help(string.lower)
Help on built-in function lower:
lower() method of builtins.str instance
    Return a copy of the string
converted to lowercase.
```

Assignment 1-B

Complementary Chain

Description

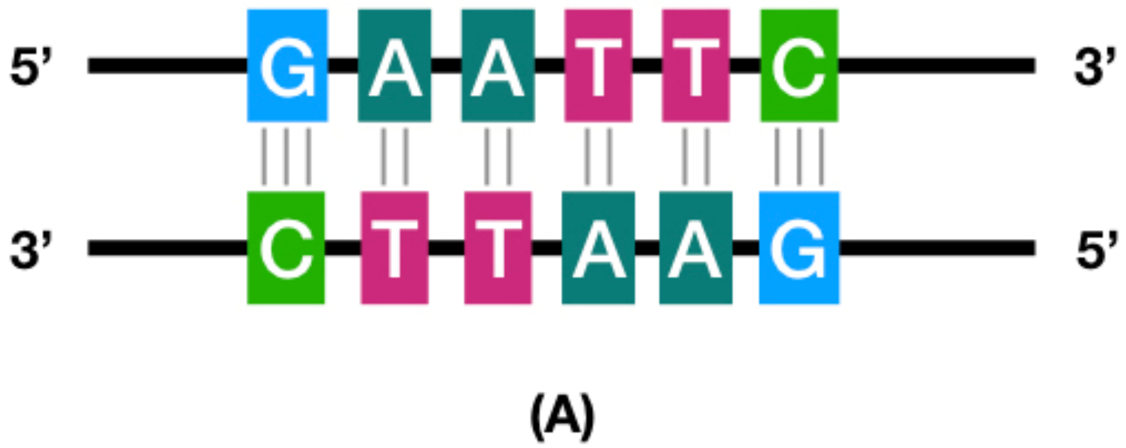
In genetics, DNA palindrome sequence refers to the specified fragment in double-stranded DNA or RNA. The order (5’-3’) of the nucleotide in this fragment is the same as the order (3’-5’) of its complementary chain.

Question

Please use the function to write the complementary chain DNA of AGGCCGAATTCGGCCT.

Note: This question has no input, and you just need to print the answer.

```
# Solve:
#dna_seq1=input()
dna_seq1='AGGCCGAATTCGGCCT'
dna_seq2=dna_seq1[::-1]
print(dna_seq2)
```



A palindromic sequence on two complementary strands

Assignment 1-C

English letters

You are given a string containing digits and English letters. Please count the number of English letters in the string.

Note: string.replace may help you filter English letters.
The input contains one line, the given string.

Input #1	Output #1
sustech2022	7
ILovePythonProgramming	22
20220905	0

GUARANTEE: the input string only contains digits (0~9), as well as English letters in upper case (A~Z) and lower cases (a~z)

```python
# Solve:
x = input()
x = x.replace('0','')
x = x.replace('1','')
x = x.replace('2','')
x = x.replace('3','')
x = x.replace('4','')
x = x.replace('5','')
x = x.replace('6','')
x = x.replace('7','')
x = x.replace('8','')
x = x.replace('9','')
print(len(x))
```

```python
# Solve:
#x = input()
x = "SUSTECH2022"
for i in x:
    if i.isdigit():
        x = x.replace(i, '')
print(len(x))
```

```
>>> help(string.isdigit)
isdigit() method of builtins.str instance
    Return True if the string is a digit string, False
otherwise.
    A string is a digit string if all characters in the
string are digits and there is at least one character in
the string.
```

```
>>> help(string.replace)
replace(old, new, count=-1, /) method of
builtins.str instance
    Return a copy with all occurrences of
substring old replaced by new.
    count: maximum number of occurrences to
replace. -1 (the default value) means replace all
occurrences.
```

Assignment 1-D

SUSTech Southern University of Science and Technology

English letters

You are given a string containing digits and English letters. Please count the number of English letters in the string.

Note: string.replace may help you filter English letters.
The input contains one line, the given string.

GUARANTEE: the input string only contains digits (0~9), as well as English letters in upper case (A~Z) and lower cases (a~z)

Input #1	Output #1
2022,9,18	9 18, 2022
2020,10,3	10 3, 2020

```
# Solve:
date_string = input()
date_list   = date_string.split(',')
print(f'{date_list[1]} {date_list[2]}, {date_list[0]}')
```

```
>>> help(string.split)
split(sep=None, maxsplit=-1) method of
builtins.str instance
    Return a list of the words in the
string, using sep as the delimiter string.
    sep: the delimiter according which to
split the string. None (the default value)
means split according to any whitespace,
and discard empty strings from the result.
    maxsplit: maximum number of splits to
do. -1 (the default value) means no limit.
```

Assignment 1-E

Sort Numbers

Description

Given an integer list of length 4, you should sort these numbers from max to min.

Input #1	Output #1
149 148 150 120	150 149 148 120
23 5434 1231 0	5434 1231 23 0

```
>>> help(sorted)
sorted(iterable, /, *, key=None,
reverse=False)
    Return a new list containing all items
from the iterable in ascending order.
    A custom key function can be supplied to
customize the sort order, and the reverse
flag can be set to request the result in
descending order.
```

```
# Solve:

input_number = input()
input_number_list = input_number.split()
sorted_input_number_list = sorted(input_number_list,reverse=True)
print(" ".join(sorted_input_number_list))
```

Assignment 1-F

Move Zeroes

Description: Given an integer list of length 5, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Input #1	Output #1
0 1 0 3 12	1 3 12 0 0
0 23 3 2 0	23 3 2 0 0

```
# Solve:
nums = input().split()
nums.append("0")
nums.remove("0")
nums.append("0")
nums.remove("0")
nums.append("0")
nums.remove("0")
nums.append("0")
nums.remove("0")
print(" ".join(nums))
```

```
# Solve:
nums = input().split()
for i in range(nums.count('0')):
    nums.remove('0')
    nums.append('0')
print(" ".join(nums))
```

```
>>> help(list.append)
append(self, object, /)
    Append object to the end of the list.
```

```
>>> help(list.remove)
remove(self, value, /)
    Remove first occurrence of value.
    Raises ValueError if the value is not present.
```

```
>>> help(list.count)
count(self, value, /)
    Return number of occurrences of value.
```

```
>>> ?range
Init signature: range(self, /, *args, **kwargs)
Docstring:
range([start,] stop[, step]) -> range object

Return an object that produces a sequence of integers
from start (inclusive) to stop (exclusive) by step.
range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted!  range(4)
produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4
elements.
When step is given, it specifies the increment (or
decrement).
```

Assignment 1-G

Word Game

Seaflower and rabbit are good friends and both of them are boring. One day, rabbit comes up with an idea that they can play a simple word game to kill the time. The rule of the game is that initially there are three words in a list. Players should add "ly"(without quote) to all of the words. Finally the words should be sorted by alphabetical order, and change the second word into "satori". After several turns, seaflower thinks the game is so boring that she does not want to play the game by herself, so she hopes you to write a program for her.

Example

The input includes three lines, each line is a word in the initial list
Output the final answer of the game.

Input #1	Output #1
rabbit is silly	['isly', 'satori', 'sillyly']

```
# Solve:
word_list = []
cur = input()
cur = cur + "ly"
word_list.append(cur)
cur = input()
cur = cur + "ly"
word_list.append(cur)
cur = input()
cur = cur + "ly"
word_list.append(cur)
word_list.sort()
word_list[1] = 'satori'
print(word_list)
```

```
>>> help(list.sort)
sort(self, /, *, key=None, reverse=False)
    Sort the list in ascending order and return None.
        The sort is in-place (i.e. the list itself is
modified) and stable (i.e. the order of two equal
elements is maintained).
    If a key function is given, apply it once to each
list item and sort them, ascending or descending,
according to their function values.
    The reverse flag can be set to sort in descending
order.
```

Assignment 1-H

SUSTech Southern University of Science and Technology

Temperature Conversion

Description

Fahrenheit and centigrade are two temperature scales in use today. The centigrade scale, which is also called the Celsius scale, was developed by Swedish astronomer Andres Celsius. In the centigrade scale, water freezes at 0 degrees and boils at 100 degrees. The Fahrenheit scale was developed by the German physicist Daniel Gabriel Fahrenheit. In the Fahrenheit scale, water freezes at 32 degrees and boils at 212 degrees.

The centigrade to Fahrenheit conversion formula is: $C=[(F-32)\times5]/9$ where F is the Fahrenheit temperature and C is the Centigrade temperature.

Request: Write a Python function to convert Fahrenheit and Celsius scale. The result retains **4** significant digits.

Note:

The `float()` method can return a floating point number from a number or a string.

Input #1	Output #1
212	100.0
97.56	36.42

```
# Solve:
Fahren_intput= float(input())
precision = 4
Cel_result = (Fahren_intput -32)*5/9
print(f'{Cel_result:.{precision}}')
```

f-string format:

- Using *variable_name* along with either *width* or *precision* values should be separated by a :
- *Precision* refers to the total number of digits that will be displayed in a number
- By default, strings are left-justified and numbers are right-justified

Assignment 1-I

SUSTech Southern University of Science and Technology

GC Content

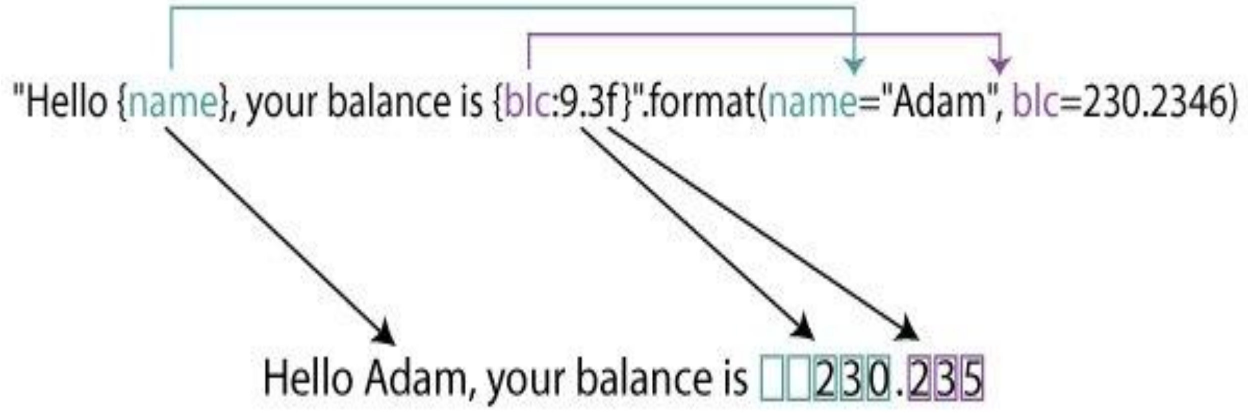
Description

Given a DNA sequence, compute the GC content (%).
The result is rounded to **2** decimal places.

Input #1	Output #1
ATTGGC	50.00%
ATT	0.00%

```
# Solve:

dna_seq=input()
GC_content=(dna_seq.count('G')+ \
dna_seq.count('C'))/len(dna_seq)

print("{0:.2f}%".format(GC_content*100))
```

```
>>> help(string.format)
format(...)
    S.format(*args, **kwargs) -> str
        Return a formatted version of S, using substitutions
from args and kwargs.
    The substitutions are identified by braces ('{' and '}').
```


"Hello {name}, your balance is {blc:9.3f}".format(name="Adam", blc=230.2346)
Hello Adam, your balance is 230.235

- f specifies the format is dealing with a float number. If not correctly specified, it will give out an error.
- The part before the "." (9) specifies the minimum width/padding the number (230.2346) can take. In this case, 230.2346 is allotted a minimum of 9 places including the ".".
If no alignment option is specified, it is aligned to the right of the remaining spaces. (For strings, it is aligned to the left.)
- The part after the "." (3) truncates the decimal part (2346) upto the given number. In this case, 2346 is truncated after 3 places.
Remaining numbers (46) is rounded off outputting 235.

Assignment 2-A

Student Names

Suppose you were a TA (Teaching Assistant) of a course in SUSTech. This course contains two examinations: midterm exam and final exam. The scores of students are already recorded. Please find those who got the same score in the midterm and final, and print out their names sorted by alphabetical order.

The input contains four lines:

- The list of students who attended the midterm exam,
- The list of midterm scores of the students who attended,
- The list of students who attended the final exam
- The list of final scores of the students who attended.

Input #1	Output #1
amy bob carol sam 95 30 70 85 bob amy sam carol 85 95 85 20	amy sam

It is guaranteed that everyone in the class has attended the midterm and the final exam, while the order of the names may differ. All of the input names are in lower cases, and all the scores are integers from 0 to 100.

The output should contain several lines. In each line, print the names of the students who satisfy the requirements.

```
# Solve:
a = input()
b = input()
c = input()
d = input()
mid_name = a.split()
mid_score = list(map(int,b.split()))
#mid_score = [int(i) for i in b.split()]
final_name = c.split()
final_score = list(map(int,d.split()))
mid_set = set(zip(mid_name,mid_score))
final_set = set(zip(final_name,final_score))
l = list(mid_set & final_set)
l.sort()
for ll in l:
    print(ll[0])
```

```
mid_set: {('amy', 95), ('bob', 30), ('carol', 70), ('sam', 85)}
final_set: {('amy', 95), ('bob', 85), ('carol', 20), ('sam', 85)}
l: [('amy', 95), ('sam', 85)]
```

```
>>> help(zip)
zip(*iterables) --> A zip object yielding tuples until an input
is exhausted.
  >>> list(zip('abcdefg', range(3), range(4)))
  [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]

 The zip object yields n-length tuples, where n is the number of
iterables passed as positional arguments to zip().  The i-th
element in every tuple comes from the i-th iterable argument to
zip().  This continues until the shortest argument is exhausted.
```

Assignment 2-B

Description

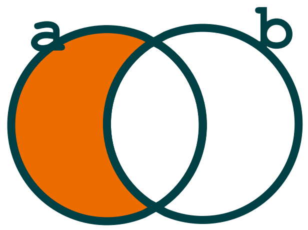
Amino acids are molecules that combine to form proteins. Amino acids and proteins are the building blocks of life. Amino acids can be represented as a single letter, for example, G is Glycine and P is Proline.
Now there is a amino acid `sequenceA: MDDDIAALVVDNGSGMCKAGF`. Please calculate the amino acid composition similarity between sequenceA and the amino acid sequence you read from input which is thought to be sequenceB. The result is rounded to **4** decimal places.
The similarity is calculated as

All types of amino acids appeared in two sequences / All types of amino acids shared by two sequences

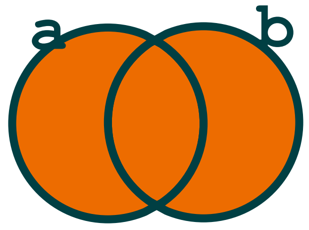
For example, sequence AALGG and seqeunce VVDLG share two amino acids: L and G. All types of amino acids appeared in these two seqeunces are: A, L, G, V, D. Thus the similarity is 2/5 = 0.40002/5=0.4000.

Input #1	Output #1
ILTLKYPIEHGIVTNWDDME	0.4444
MIAALVAGFFF	0.5833

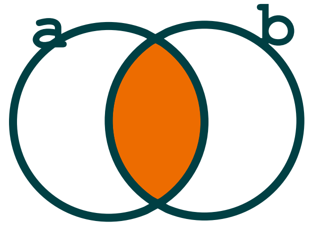
```
sequenceA='MDDDIAALVVDNGSGMCKAGF'

sequenceB=input()

similarity = len(set(sequenceA)& set(sequenceB)) \
/ len(set(sequenceA)|set(sequenceB))

print("{0:.4f}".format(similarity))
```



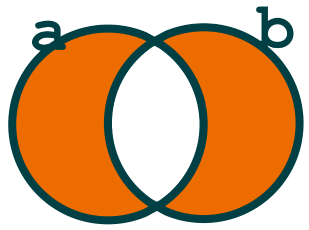
Difference
a-b



Union
a|b (A ∪ B)



Intersection
a&b (A ∩ B)



Symmetric difference
a^b (A Δ B)

Assignment 2-C

SUSTech Southern University of Science and Technology

Sort Dictionary

Description

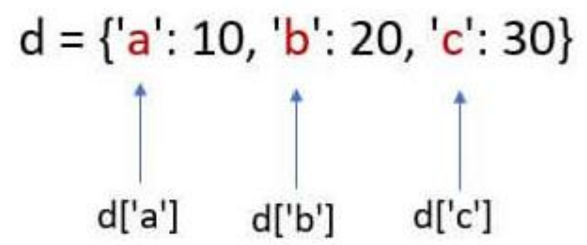
Given two lines of input, each with **4** numbers. The **1st** line is the key and the **2nd** line is the value. Please output the **2nd** line in ascending order of the **1st** line. We guarantee that the numbers in the first line are all different.

Input #1	Output #1
23 34 12 546 1 2 3 4	3 1 2 4
897 231 -213 123 4 3 2 1	2 1 3 4

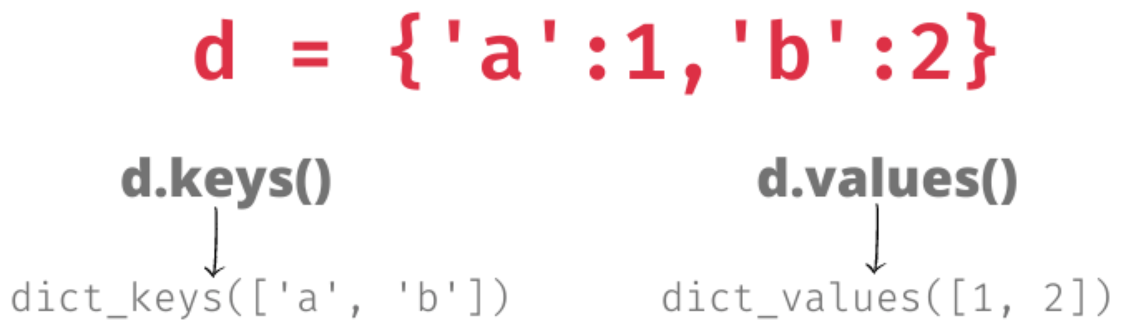
```
a = list(map(int, input().split()))
b = input().split()
c = dict()
c[a[0]]=b[0]
c[a[1]]=b[1]
c[a[2]]=b[2]
c[a[3]]=b[3]
d = sorted(c.keys())
print(c[d[0]],c[d[1]],c[d[2]],c[d[3]])
```

```
for k, v in zip(a, b):
    c[k] = v
```

Unordered collections of unique values stored in (Key-Value) pairs.



- ✓ **Unordered**: The items in dict are stored without any index value
- ✓ **Unique**: Keys in dictionaries should be Unique
- ✓ **Mutable**: We can add/Modify/Remove key-value after the creation



Assignment 2-D

Report card

Description

If the report card of the four students in the last test were as follows:

```
student = ("12210001", "12210002",
           "12210003", "12210004")
score = ("90", "99", "85", "97")
level = ("A+", "A+", "A", "A+")
```

Now, you need to calculate a classmate's GPA based on score and report that classmate's report card.

```
60 = 1.0
61 = 1.1
70 = 2.0
90 = 4.0
"> 90" = 4.0
```

Input #1	Output #1
12210001	('12210001', '90', '4.0', 'A+')
12210002	('12210002', '99', '4.0', 'A+')

```
>>> help(list.insert)

insert(self, index, object, /)
    Insert object before index.
```

```
student = ("12210001", "12210002",
           "12210003", "12210004")
score = ("90", "99", "85", "97")
level = ("A+", "A+", "A", "A+")
student_score_lst = list(zip(student, score,
level))

curr_id = input() #'12210001'
curr_idx = student.index(curr_id)
curr_score_lst =
list(student_score_lst[curr_idx])
if int(score[curr_idx]) < 90:
    GPA = str(1.0+(int(score[curr_idx])-60)*0.1)
else:
    GPA = str(4.0)
curr_score_lst.insert(2, GPA)
curr_score_tpl = tuple(curr_score_lst)
print(curr_score_tpl)
```

```
Student_score_lst:
[('12210001', '90', 'A+'),
 ('12210002', '99', 'A+'),
 ('12210003', '85', 'A'),
 ('12210004', '97', 'A+')]
```

```
>>> help(list.index)
index(self, value, start=0,
stop=9223372036854775807, /)
    Return the first index of value. Raises
ValueError if the value is not present.
```

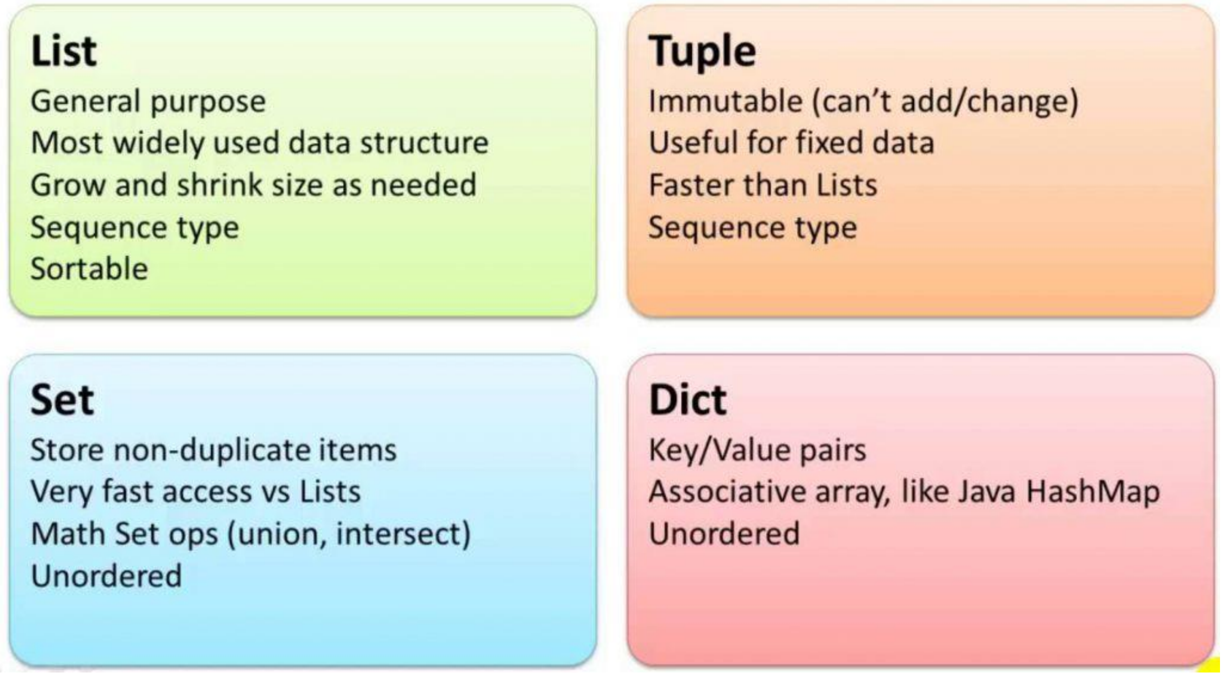
Assignment 2-E

Class Attendance

You were told to assist the teacher to summarize the attendance situation of CS112 2022 Fall.
In general, students write their names on one A4 paper.
However, some naughty students tried to confuse you by recording their names twice or more times. And their names can be anywhere in the list in any order.
Fortunately, you are a skilled programmer, especially for Python language.
Given a series of student names (splited by ','), your task is to print out the total number of students (remember some names are duplicated and should be counted only once) and then list the names in alphabetical order.

Input #1	Output #1
Jacob,Mason,William,Mason	3 Jacob Mason William

```
name_list = input().split(',')
name_set = set(name_list)
name_list = list(name_set)
num_name = len(name_list)
print(num_name)
sorted_name_list = sorted(name_list)
for name in sorted_name_list:
    print(name)
```



List
General purpose
Most widely used data structure
Grow and shrink size as needed
Sequence type
Sortable

Tuple
Immutable (can't add/change)
Useful for fixed data
Faster than Lists
Sequence type

Set
Store non-duplicate items
Very fast access vs Lists
Math Set ops (union, intersect)
Unordered

Dict
Key/Value pairs
Associative array, like Java HashMap
Unordered

Assignment 2-F

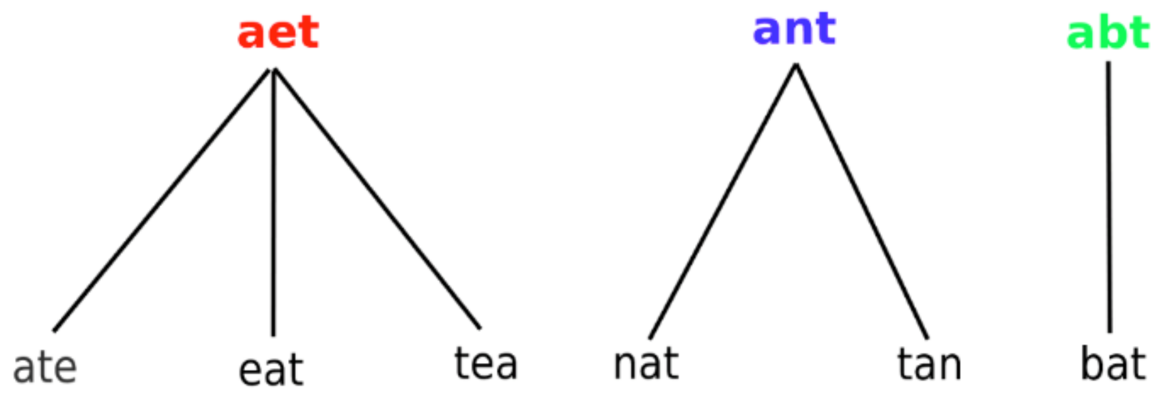
Group Anagrams And Count

Given an array of strings strs, group the anagrams together and give the number of groups.
An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.
For example, “abcd” and “dabc” are anagram of each other.

Note:

- Strings always contain only 6 words
- Strings only consists of lowercase English letters.

Input #1	Output #1
eat tea tan ate nat bat	3
cat tta tcc python thonpy bat	5



```
ans = {}
input_str = "eat tea tan ate nat bat" #input()
input_str = "cat tta tcc python thonpy bat"
str_lst = input_str.strip().split()
for w in str_lst:
    ws_lst = sorted(w)
    ws_str = "".join(ws_lst)
    if ws_str in ans:
        ans[ws_str] = ans[ws_str] + 1
    else:
        ans[ws_str] = 1
print(len(ans))
```

The key can be a string or tuple, but not a list

```
>>> sorted('tae')
['t', 'e', 'a']
>>> sorted('ate')
['t', 'e', 'a']
```

```
>>> help(string.join)
join(iterable, /) method of builtins.str instance
    Concatenate any number of strings.
    The string whose method is called is inserted
in between each given string.
    The result is returned as a new string.
    Example: '.'.join(['ab', 'pq', 'rs']) ->
'ab.pq.rs'
```

Assignment 2-G

Counting

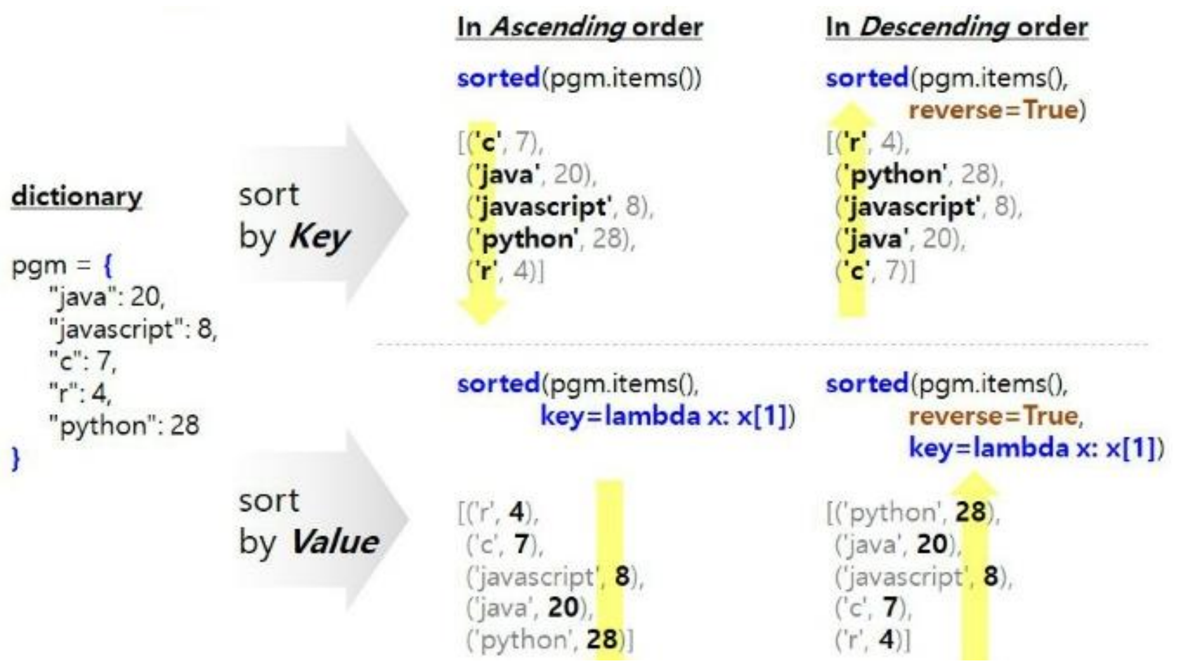
Satori is a girl who likes statistics. To make the input number in order, she decides to put them in a dictionary. The key is a number, and the value is the number of times that number appears. The dictionary should be sorted by the keys in ascending order. The keys should be treated as strings when sorting.

Example

Input consists of six lines of numbers. Please print each key and value pair in a line, separate them with ": ".

Input #1	Output #1
2 3 4 5 6 3	2: 1 3: 2 4: 1 5: 1 6: 1

```
ans = {}
for i in range(6):
    num = input()
    if num in ans:
        ans[num] = ans[num] + 1
    else:
        ans[num] = 1

ans_order=dict(sorted(ans.items()))

for key, val in ans_order.items():
    print('{key}: {value}'.format(key = key, value = val))
```



```
pgm = {
    "java": 20,
    "javascript": 8,
    "c": 7,
    "r": 4,
    "python": 28
}
```

```
sorted(pgm.items())

[('c', 7),
 ('java', 20),
 ('javascript', 8),
 ('python', 28),
 ('r', 4)]
```

```
sorted(pgm.items(),
       reverse=True)

[('r', 4),
 ('python', 28),
 ('javascript', 8),
 ('java', 20),
 ('c', 7)]
```

```
sorted(pgm.items(),
       key=lambda x: x[1])

[('r', 4),
 ('c', 7),
 ('javascript', 8),
 ('java', 20),
 ('python', 28)]
```

```
sorted(pgm.items(),
       reverse=True,
       key=lambda x: x[1])

[('python', 28),
 ('java', 20),
 ('javascript', 8),
 ('c', 7),
 ('r', 4)]
```

Quiz

Quiz 1-A

Southern University of Science and Technology

List Shift

Description

Given an integer a and an integer n. Shift a to the right by the specified number of places n, and the shifted places are filled to the left vacancy.

Input #1	Output #1
2312 1	2231
23100 2	00231
872 4	287

```
a = list(input())
n = int(input())
n = n%len(a)

a = a[len(a)-n:]+a[:len(a)-n]
print("".join(a))
```

Quiz 1-B

Cheapest Store

You are going to buy a laptop, walking around the market. Each of the store that you have visited have offered different prices for the laptop. Suppose you visited ***THREE*** stores, please print the name of the store that offers the lowest price.

The input contains three lines. Each line writes down the name of the store and the price it offered. The prices are all integers, and all the three prices are different.

The output contains one line, where you print the store name.

Input #1	Output #1
Store1 2000 Store2 1900 Store3 2100	Store2

Hint: You can store the data as a *list of tuples*. Such list can be sorted according to the first element in each tuple.

```
a = input()
b = input()
c = input()
l = []
l.append(tuple(a.split()[::-1]))
l.append(tuple(b.split()[::-1]))
l.append(tuple(c.split()[::-1]))
l.sort()
print(l[0][1])
```

Quiz 1-C

SUSTech Southern University of Science and Technology

Tour Plan

Description

Assume you are planning a tour around the world. You have created plenty of tour plans, for example, traveling with city order: "city1 → city2 → city3 → city1". However, you are annoyed at calculating the total tour cost (total distance).
So, you decide to write a Python program that automatically calculates the distance, taking the city list as input. With the help of your program, you are happy now that you do not need to do it by hand anymore.

Notes:

- The input contains 2 lines representing the coordinates (**float number**) of **3 cities**:
  - E.g., (0,0),(3,4),(6,0)(0,0),(3,4),(6,0).
  - x-axis value of each city: e.g., "0 3 6".
  - y-axis value of each city: e.g., "0 4 0".
- **Your task is to calculate the total distances along the cities and finally back to the first city.**
  - $D = \sqrt{(3-0)^2+(4-0)^2} + \sqrt{(6-3)^2+(0-4)^2} + \sqrt{(0-6)^2+(0-0)^2} = 5+5+6 = 16.$
  - In python, the square root ($\sqrt{\cdot}$) operation can be implemented using `**0.5`. E.g., `4**0.5==2`.
- Please **round** the output to the integer.

```
x = input().split()
y = input().split()
x = [float(i) for i in x]
y = [float(i) for i in y]
distance = 0
distance += ((x[1]-x[0])**2 + (y[1]-y[0])**2)**0.5
distance += ((x[2]-x[1])**2 + (y[2]-y[1])**2)**0.5
distance += ((x[0]-x[2])**2 + (y[0]-y[2])**2)**0.5

print(round(distance))
```

Input #1	Output #1
0 3 6 0 4 0	16
2.24 3.68 3.84 3.80 3.99 3.82	3