

CS112

Introduction to Python Programming

互助课堂Session 03: Final Review1

GAO Jianxing

Modified based on the PPR from Prof.HOU
Shengwei and YI Xiang

Schedule



Week	Lecture	Lab	Quiz	Assignments
1	Course Intro (1) + Python Intro (1)	IDE, Data Types, Operators		
2	Strings (1) + Lists (1)	Strings, Lists		
3	Strings & Lists (0.5) + Dictionaries (1.5)	Strings, Lists, Dictionaries		A1
4	Tuples (1) + Sets (1)	Tuples, Sets		
5	Flow Control (2)	Flow Control	Quiz1	A2
6	Functions (2)	Functions		
7	Input & Output (2)	Input & Output		A3
8	Midterm Recapitulation	Midterm Recapitulation		
9	NumPy Arrays (2)	NumPy Arrays	Session1	A4
10	NumPy (1) & SciPy (1)	NumPy & SciPy	Quiz2	
11	Pandas (2)	Pandas	Session2	A5
12	Data Visualization (2)	Data Visualization		
13	Data Visualization (2)	Data Visualization	Session3	A6
14	Object-Oriented Programming (OOP) (2)	OOP	Quiz3	
15	Object-Oriented Programming (2)	OOP	Session4	A7
16	Summary (2)	Summary		

- Python 基本概念介绍
- 基本数据类型
- 控制流
- 函数
- 输入输出
- Numpy
- Pandas
- 类与对象
- Plotting

- Python 基本概念介绍
- 基本数据类型
- 控制流
- 函数
- 输入输出
- Numpy
- Pandas
- 类与对象
- Plotting
- 机器码、汇编语言和高级语言
- 编译型语言 and 解释型语言
- Keywords
- Python 的编写规范



• Machine Code 机器码

- low-level computer language
- can be directly understandable by a computer 计算机能读懂
- all programs must be converted to machine code before they can be run
- Can run and execute very fast 编程语言须转换成机器码
- Almost impossible to read; hard to maintain and debug 但是人类难读懂



• Assembly Language 汇编语言

- Human readable notation for the machine language, but still difficult for human to read 人类可读懂，但是晦涩
- Computer cannot understand assembly language; assembler must be used to convert assembly language to machine language 计算机不能读懂
- Machine-dependent; difficult for portability 换一台电脑就不能运行了

• High-level Language 高级语言

High-level Language



- Close to human language
- Platform independent 同一段代码在任何设备都可运行
- Easier to modify, faster to write code and debug
- Must be translated into machine language before it can run (translation is done using either **compiler** or **interpreter**)
- 注意区分解释执行语言和编译执行语言



- An **interpreter** reads source code one statement at a time, and translates it into machine language and **executes** it
- A **compiler** transforms high-level source code into a low-level machine language – **compilation**
- Compiled languages are generally faster than interpreted languages

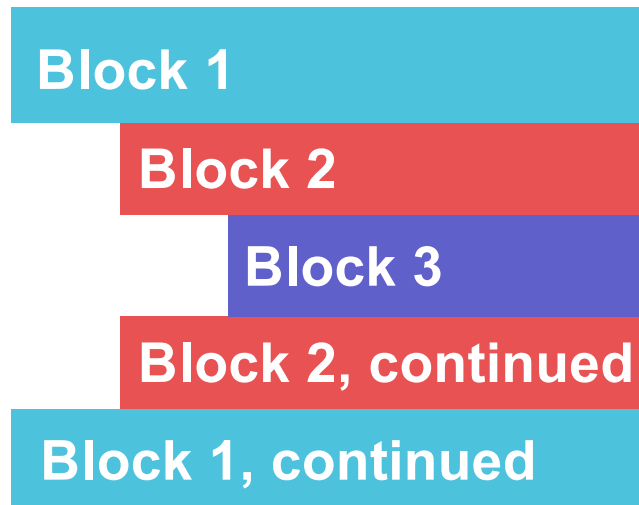
Operators

- Arithmetic Operators 算术运算符
 - + - * / // (向下整除) % (取余) **
- Assignment Operators 赋值运算符
 - + =, - =, * =, and / = **Compound**
 - % =, ** =, and // = **assignment operators**
- Comparison Operators 比较运算符
- Logical Operators 逻辑运算符
 - and, or, not
- Precedence and Associativity 优先/组合

Python doesn't support Autoincrement (++) and Autodecrement (--) operators

Indentation缩进

- Python code is structured through indentation
- In Python indentation is a requirement and not a matter of style
- Any statements written under another statement with the same indentation is interpreted to belong to the same code block
- Usually, **four whitespaces** are used for indentation and are preferred over tabs



Indenting some lines by 4 spaces and other lines by a <tab> character will produce an error

Identifier 标识符



- An identifier is a name given to a variable, function, class or module.
- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_)
- Identifiers **must** start with a **letter** or an **underscore**, **cannot start with a digit**
myCountry, other_1, good_morning, _name

1plus, 5_



- Spaces are not allowed in Identifiers 单个标识符不能有空格

desk height = 2.



desk_height = 2.



- Special symbols like !, @, #, \$, % cannot be used in identifiers

\$money = 10



- Identifiers in Python are **case-sensitive** 大小写敏感

height is different from Height

- **Keywords** cannot be used as identifiers 关键词不能用作标识符



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	

- Python 基本概念介绍
- 基本数据类型
- 控制流
- 函数
- 输入输出
- Numpy
- Pandas
- 类与对象
- Plotting

Python 中的六大基本类型

- 数字型
 - 整数
 - 浮点数
 - 布尔
 - 复数
- 字符串
- 列表
- 字典
- 集合
- 元组(tuple)

- Python 中的六大基本类型 对象

Data types 数据类型



- Integers:

```
>>> type(2)
<class 'int'>
```

- Floating numbers:

```
>>> type(2.)
<class 'float'>
```

- Complex numbers:

```
>>> type(2+3j)
<class 'complex'>
```

- Boolean: True or False

```
>>> type(True)
<class 'bool'>
```

- Strings:

A sequence of one or more characters; includes letters, numbers, and other types of characters such as spaces
Single quotes or double quotes can be used to represent strings

```
>>> type('Hello, World!')
<class 'str'>
>>> type("Hello, World!")
<class 'str'>
```

None: None is used to represent the absence of a value

```
>>> money = None
>>> type(money)
<class 'NoneType'>
```

```
>>> type(4/2)
<class 'float'>
```

integer * float → float
float * complex → complex

Python promotes the result to the most complicated type of the inputs

Type conversions 类型转换

- int() Function
- float() Function
- str() Function
- chr() Function
- complex() Function



```
[2]: help(id)
```

```
Help on built-in function id in module builtins:
```

```
id(obj, /)
```

```
Return the identity of an object.
```

```
This is guaranteed to be unique among simultaneously existing objects.  
(CPython uses the object's memory address.)
```

```
intvar = 1
```

```
floatvar = 0.1
```

```
boolvar = False
```

```
complexvar = 1 + 3j
```

```
listvar = ['a', 'b']
```

```
tuplevar = (0, 1)
```

```
dictvar = {'yx': 'yxx'}
```

```
strvar = 'test_str'
```

```
print(type(intvar), id(intvar))
```

```
print(type(floatvar), id(floatvar))
```

```
print(type(boolvar), id(boolvar))
```

```
print(type(complexvar), id(complexvar))
```

```
print(type(listvar), id(listvar))
```

```
print(type(tuplevar), id(tuplevar))
```

```
print(type(dictvar), id(dictvar))
```

```
print(type(strvar), id(strvar))
```

Mutable and Immutable



- **不可变对象**：不支持原地修改。一旦修改，便一定是创建了一个新对象。
- **可变对象**：支持原地（in-place）修改，且不改变原对象。同时也支持形如不可变对象的修改方式。

```
listvar[2] = 'b'  
strvar[2] = 'b'
```

```
-----  
TypeError                                Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_11116\2023988296.py in <module>  
      1 listvar[2] = 'b'  
----> 2 strvar[2] = 'b'  
  
TypeError: 'str' object does not support item assignment
```

数据类型	是否可变?
数字型	不可变
字符串	不可变
元组	不可变
列表	可变
集合	可变
字典	可变

Mutable and Immutable

**intvar**

Identity:
0xdeadbeef
Type: int
Value: 123

Identity:
0xcafecafe
Type: int
Value: 144

`intvar = 123`
`intvar = 144`

listvar

Identity:
0xdeadbeef
Type: list
Value: [1, 2, 3, 4]

`listvar = [1, 2, 3]`
`listvar.append(4)`

- Python 基本概念介绍
- 基本数据类型
- 控制流
- 函数
- 输入输出
- Numpy
- Pandas
- 类与对象
- Plotting

Python 中的六大基本类型

- 数字型
 - 整数
 - 浮点数
 - 布尔
 - 复数
- 字符串
- 列表
- 字典
- 集合
- 元组(tuple)

- Python 中的六大基本类型 对象

String

- Strings can be concatenated using the “+” operator 字符串的拼接
- Strings can also be repeated using the “*” operator 字符串的复制
- **in** and **not in**: check for the presence of a string in another string
- Comparison: **>**, **<**, **<=**, **>=**, **==**, **!=**:
 - based on the ASCII value
- 字符串的长度 **len()** Number of characters in a string
- 统计（聚合）： **min()** and **max()** return a character having the highest and lowest ASCII value **max("axel")** 'x'

String indexing 索引

- Each of the string's character corresponds to an index number starting from 0; square brackets are used to perform indexing in a string:

```
>>> phrase = "be yourself"
```

```
>>> len(phrase)
```

```
11
```

```
>>> phrase[0]
```

```
'b'
```

```
>>> phrase[10]
```

```
'f'
```

```
>>> phrase[11]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: string index out of range
```

```
>>> phrase[-1]
```

```
'f'
```

```
>>> phrase[-2]
```

```
'l'
```

**Python's index
starts from zero**

String slicing 切片



- 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:

```
>>> phrase = "be yourself"
>>> phrase[0:4] 'be y'
>>> phrase[:4] 'be y'
>>> phrase[6:] 'rself'
>>> phrase[:] 'be yourself'
>>> phrase[0:9:3] 'byr'
>>> phrase[0::3] 'byrl'
```

```
>>> phrase[4:4]
''
>>> phrase[4:20]
'ourself'
>>> phrase[-4:-1]
'sel'
>>> phrase[4:-1]
'ourself'
```

- 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 one

String slicing and split



- Strings can be joined with the `join()` function:

```
>>> date_of_birth = ["17", "09", "1950"]  
>>> ":".join(date_of_birth)  
'17:09:1950'
```

```
>>> social_app = ["instagram", "is", "a", "photo", "sharing",  
"application"]  
>>> " ".join(social_app)  
'instagram is a photo sharing application'
```

```
>>> "123".join("amy")  
'a123m123y'
```

- Strings can be split with the `split()` function

```
>>> inventors = "edison, tesla, marconi, newton"  
>>> inventors.split(",")  
['edison', ' tesla', ' marconi', ' newton']
```

```
>>> watches = "rolex hublot cartier omega"  
>>> watches.split()  
['rolex', 'hublot', 'cartier', 'omega']
```

String method examples



```
>>> "sailors".isalpha()  
True
```

```
>>> "2018".isdigit()  
True
```

```
>>> "Fact".islower()  
False
```

```
>>> "TSAR BOMA".isupper()  
True
```

```
>>> "galapagos".upper()  
'GALAPAGOS'
```

```
>>> "TSAR BOMA".lower()  
'tsar boma'
```

```
>>> "gAlPas".capitalize()  
'Galpas'
```

```
>>> "cucumber".find("cu")  
0
```

```
>>> "cucumber".find("um")  
3
```

```
>>> "alpha".count("a")  
2
```

```
>>> "Centennial Light".swapcase()  
'cENTENNIAL LIGHT'
```

Format strings

- Use `str.format()` method if you need to insert the value of a variable, expression or an object into another string

```
>>> country = "China"
>>> print("I live in {}".format(country))
I live in China
>>> a = 10, b = 20
>>> print("The values of a is {} and b is {}".format(a, b))
>>> print("The values of b is {} and a is {}".format(a, b))
```

Index value starts from zero

- f-string format:

```
f"string_statements {variable_name [: {width}.{precision}]}"
```

```
>>> width = 10
>>> precision = 5
>>> value = 12.34567
>>> f'result: {value:{width}.{precision}}'
'result:      12.346'
>>> f'result: {value:{width}}'
'result:    12.34567'
>>> f'result: {value:.{precision}}'
'result: 12.346'
```

- 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

Escape Sequences and Raw String



- Escape sequences are a combination of a backslash (\) followed by either a letter or a combination of letters and digits

\\	Inserts a Backslash character in the string
\'	Inserts a Single Quote character in the string
\"	Inserts a Double Quote character in the string
\n	Inserts a New Line in the string
\t	Inserts a Tab in the string
\b	Inserts a Backspace in the string

\ Break a Line into Multiple lines while ensuring the continuation

```
>>> print("You can break \
... single line to \
... multiple lines")
You can break single line to multiple lines
```

```
>>> print("a"c")
File "<stdin>", line 1
    print("a"c")
^SyntaxError: invalid syntax
>>> print("a\"c")
a"c
```

```
>>> print(r"he asked, \"What\"s the best way to code
the snake game?\" She answered, \"In *python* script\"")
```

```
he asked, \"What\"s the best way to code the snake
game?\" She answered, \"In *python* script\"
```

- 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

- Python 基本概念介绍
- 基本数据类型
- 控制流
- 函数
- 输入输出
- Numpy
- Pandas
- 类与对象
- Plotting

Python 中的六大基本类型

- 数字型
 - 整数
 - 浮点数
 - 布尔
 - 复数
- 字符串
- 列表
- 字典
- 集合
- 元组 (tuple)
- **Python 中的六大基本类型 对象**

• List

```
>>> b = [7., "girl", 2+0j, "horse", 21]
```

- + * in, not in, len()
- List indexing
- List slicing
- List modification
- List sorting & aggregation
- “增删插改”
- append(), insert(), remove(), pop(), del()

• Dictionary

- Unordered, key:value pair
- dict() and modification
- len(), sorted(), pop()
- 增删查改: update(), get()

• Tuples

- ordered, immutable and heterogeneous list
- Tuple indexing and index()
- Tuple slicing
- Tuple aggregation
- zip()

```
>>> x = [1, 2, 3]      >>> list(zippered)
>>> y = [4, 5, 6, 7]  [(1, 4), (2, 5), (3, 6)]
```

• Set

- unordered collection with no duplicate items
- set() and modification
- len(), sorted(), update()
- add(), discard()
- frozenset(): immutable

List Aggregation (Description) 统计



```
>>> numbers = [1, 2, 3, 4, 5]
>>> sum(numbers)
15
>>> max(numbers)
5
>>> min(numbers)
1
>>> list_1=[1, 2, 3, 1]
>>> list_1.count(1)
2
```

List sorting 排序

```
>>> list_1 = [1, 5, 2, 3, 1]
>>> list_1.index(5)
1
```

`index()` returns the index for the given item from the start of the list

```
>>> list_2 = list_1.copy()
>>> list_2[1] = 6
>>> list_1
[1, 5, 2, 3, 1]
>>> list_2
[1, 6, 2, 3, 1]
```

`copy()` creates a new copy of the existing list

```
>>> list_1.reverse()
>>> list_1
[1, 3, 2, 5, 1]
>>> list_1.sort()
>>> list_1
[1, 1, 2, 3, 5]
```

```
p.sort(reverse=True)
sorted(p, reverse=True)
```

```
>>> zoo_sorted = sorted(zoo)
>>> zoo_sorted
['Lion', 'Tiger', 'Zebra']
>>> zoo
['Zebra', 'Tiger', 'Lion']
```

- *The `sorted()` function returns a modified copy of the list while without modifying the original list*
- *The list is sorted based on the ASCII value*

List “增删插改”

- The `append()` adds a single item to the end of the list:

```
>>> list_1 = [1,1,2,3]
>>> list_1.append(5)
>>> list_1
[1, 1, 2, 3, 5]
```

- The `remove()` searches for the first instance of the given item in the list and removes it

```
>>> list_1.remove(3)
>>> list_1
[1, 1, 2, 5]
```

- The `insert()` method inserts the item at the given index, shifting items to the right:

```
>>> list_1.insert(1,5)
>>> list_1
[1, 5, 1, 2, 5]
```

The list size changes dynamically whenever you add or remove the items.

List pop() and del Statement



- The `pop()` **removes and returns** the item **at the given index**; returns the **rightmost** item if the index is omitted 删除索引对应的项，并**返回**删除的值。 **LIFO**

```
>>> list_1 = [1, 1, 2, 5]
>>> list_1.pop(3)
5
>>> list_1
[1, 1, 2]
>>> list_1.pop()
2
>>> list_1
[1, 1]
```

```
>>> a = [5, -8, 99.99, 432, 108, 213]
>>> del a[0]
>>> a
[-8, 99.99, 432, 108, 213]
>>> del a[2:4]
>>> a
[-8, 99.99, 213]
>>> del a[: ]
>>> a
[]
```

```
>>> del a
```

```
>>> a
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
NameError: name 'a' is not defined
```

- `del` statement removes an item from a list based on its index
- The difference between `del` and `pop()` is that `del` does not return any value while `pop()` returns a value. `del`删除元素时没有返回值
- The `del` statement can also be used to remove slices from a list or clear the entire list 可用于删除切片乃至整个列表

The del Statement

- `del` statement removes an item from a list based on its index
- The difference between `del` and `pop()` is that `del` does not return any value while `pop()` returns a value
- The `del` statement can also be used to remove slices from a list or clear the entire list

```
>>> a = [5, -8, 99.99, 432, 108, 213]
>>> del a[0]
>>> a
[-8, 99.99, 432, 108, 213]
>>> del a[2:4]
>>> a
[-8, 99.99, 213]
>>> del a[:]
>>> a
[]
```

```
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

• List `>>> b = [7., "girl", 2+0j, "horse", 21]`

- `+` `*` `in`, `not in`, `len()`
- List indexing
- List slicing
- List modification
- List sorting & aggregation
- “增删插改”
- `append()`, `insert()`, `remove()`, `pop()`, `del()`

• Dictionary

- **Unordered**, key:value pair 键值对
- `dict()` and modification
- `len()`, `sorted()`, `pop()`
- 增删查改: `update()`, `get()`

• Tuples

- ordered, **immutable** and heterogeneous list
- Tuple indexing and `index()`
- Tuple slicing
- Tuple aggregation
- `zip()`

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6, 7]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
```

• Set

- **unordered collection with no duplicate items**
- `set()` and modification
- `len()`, `sorted()`, `update()`
- `add()`, `discard()`
- `frozenset()`: **immutable**

Assignment 3-H



Who is the winner

- J=11, Q=12, K=13, there is no big or small king, and the A (1) is the biggest one in the single cards.
- For card combinations, all three cards being the same is the biggest (5,5,5), followed by straights (4,5,6), pairs (4,4,5), and finally singles (4,5,7). If the combinations are the same type, **compare the numbers in weighted.**

Input

The inputs are the cards that the three players got in their hands.

Output

Judge who is the winner.

Dictionary

- A dictionary is a collection of an unordered set of **key:value** pairs, with the requirement that **the keys must be unique within a dictionary**; (一个字典中不允许存在两个相同的键)
- Dictionaries are created using **curly brackets {}**:

```
dictionary_name = {key_1:value_1, key_2:value_2,  
key_3:value_3, .....,key_n:value_n}
```

```
>>> room = {"Emma":309, "Jake":582, "Olivia":764}  
>>> type(room)  
<class 'dict'>
```

- The entries in a dictionary are separated by **commas**; (逗号分隔)
- Each entry consists of a key and a value; each key and its value are separated by a **colon**; (冒号)
- **Dictionaries are indexed by keys**, and the entries can be accessed by keys:

```
dictionary_name[key]
```

```
>>> room["Olivia"]  
764
```

- Dictionary keys are case sensitive, and must be **immutable**! 键不可变, 但是字典可变
- Duplicate keys are not allowed in the dictionary

Dictionary creation

- Dictionaries can be built up and added to in a straightforward manner:

```
>>> d = {}
>>> d["last name"] = "Alberts"
>>> d["first name"] = "Marie"
>>> d["birthday"] = "January 27"
>>> d
{'last name': 'Alberts', 'first name': 'Marie', 'birthday': 'January 27'}
```

- In dictionaries, the order of **key:value** pairs does not matter:

```
>>> d_new = {'birthday': 'January 27', 'first name': 'Marie', 'last name': 'Alberts'}
>>> d == d_new
True
```

- Slicing in dictionaries is not allowed since they are **not ordered like lists**;
- The built-in **dict()** function is used to create dictionary. (方法2)

```
>>> numbers = dict(one=1, two=2, three=3)
>>> numbers
{'one': 1, 'two': 2, 'three': 3}
>>> new_dict = dict()
>>> new_dict
{}
```

Dictionary modification

- The syntax for modifying the value of an existing key or for adding a new **key:value** pair to a dictionary is:

```
dictionary_name[key] = value
```

- If the key is already in the dictionary, its value will be updated with the new value; if the key is not present, the new **key:value** pair will be added to the dictionary.

```
>>> d = {'last name': 'Alberts', 'first name': 'Marie', 'birthday': 'January 27'}
```

```
>>> d["last name"] = "Johnson"
```

Dictionaries are mutable

```
>>> d
```

```
{'last name': 'Johnson', 'first name': 'Marie', 'birthday': 'January 27'}
```

```
>>> d["phone number"] = 5053101021
```

```
>>> d
```

```
{'last name': 'Johnson', 'first name': 'Marie', 'birthday': 'January 27',  
'phone number': 5053101000}
```

Dictionary “增删查改”



- `.get()` returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to `None`, so that this method never raises a **KeyError**

```
>>> numbers = {'one': 1, 'two': 2, 'three': 3}
>>> numbers.get("three")
3
>>> print(numbers.get("four"))
None
>>> numbers.get("four", 4)
4
```

```
In [1]: numbers['four']
KeyError                                Traceback (most recent call last)
<ipython-input-6-175332202980> in <module>
----> 1 numbers['four']
```

KeyError: 'four'

```
In [2]: numbers.get?
Signature: numbers.get(key, default=None, /)
Docstring: Return the value for key if key is in the dictionary, else default.
Type:      builtin_function_or_method
```

- A list of all keys or values of a dictionary can be obtained by the dictionary name followed by `.keys()` or `.values()`:

```
>>> d.keys()
dict_keys(['last name', 'first name', 'birthday'])
>>> d.values()
dict_values(['Alberts', 'Marie', 'January 27'])
```

Dictionary “增删查改”



- `.update()` updates the dictionary with the **key:value** pairs from other dictionary object:

```
>>> numbers.update({"four":4})
```

```
>>> numbers
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

- `.pop()` removes the key from the dictionary and returns its value

```
>>> numbers.pop("four")
```

```
4
```

```
>>> numbers
```

```
{'one': 1, 'two': 2, 'three': 3}
```

Assignment 2-B



Input

The first line is student number m . Then m parts are given to show students' courses. Each part contains one line shown the number of courses mi and mi lines of course names and given credits.

Output

Output the most popular course and its received credits in one line separated by one space.

Example

Input #1

Copied

```
3
5
Python 33
Math 31
Art 11
EnglishII 6
Biology 19
3
Python 33
EnglishIII 34
Biology 19
3
EnglishI 12
Basketball 56
Python 32
```

Output #1

Copy

```
Python 98
```

• List `>>> b = [7., "girl", 2+0j, "horse", 21]`

- + * in, not in, len()
- List indexing
- List slicing
- List modification
- List sorting & aggregation
- “增删插改”
- append(), insert(), remove(), pop(), del()

• Dictionary

- Unordered, key:value pair
- dict() and modification
- in, not in, len(), sorted(), pop()
- 增删查改: update(), get()

• Tuples

- ordered, **immutable** and heterogeneous list
- + * in, not in, len()
- Tuple indexing and index()
- Tuple slicing
- Tuple aggregation
- zip()

• Set

- **unordered collection with no duplicate items**
- set() and modification
- len(), sorted(), update()
- add(), discard()
- frozenset(): **immutable**

The tuple() function



- The built-in `tuple()` function is used to create a tuple from an iterable object:

```
>>> tuple("sustech")  
('s', 'u', 's', 't', 'e', 'c', 'h')
```

String to tuple

```
>>> tuple(range(10))  
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Range to tuple

```
>>> tuple(['CS', '112'])  
('CS', '112')
```

List to tuple

```
>>> tuple({'a':1,'b':2})  
('a', 'b')
```

Dict to tuple

Tuple packing and unpacking



- The values 12345, 54321 and 'hello!' are packed together into a tuple
- The reverse operation of tuple packing is also possible

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

```
>>> x, y, z = t
>>> x
12345
>>> y
54321
>>> z
'hello!'
```

```
In [1]: t = 12345, 54321, 'hello!'
```

```
In [2]: t
```

```
Out[2]: (12345, 54321, 'hello!')
```

```
In [3]: x,y,z=t
```

```
In [4]: x,y = t
```

```
-----
ValueError Traceback (most recent call last)
```

```
<ipython-input-87-623752d04abb> in <module>
```

```
----> 1 x,y = t
```

```
ValueError: too many values to unpack (expected 2)
```

```
In [5]: x,y,_ = t
```

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)]
```

- List
 - `>>> b = [7., "girl", 2+0j, "horse", 21]`
 - + * in, not in, len()
 - List indexing
 - List slicing
 - List modification
 - List sorting & aggregation
 - “增删插改”
 - `append()`, `insert()`, `remove()`, `pop()`, `del()`
- Dictionary
 - Unordered, key:value pair
 - `dict()` and modification
 - `in`, `not in`, `len()`, `sorted()`, `pop()`
 - 增删查改: `update()`, `get()`
- Tuples
 - ordered, immutable and heterogeneous list
 - + * in, not in, len()
 - Tuple indexing and `index()`
 - Tuple slicing
 - Tuple aggregation
 - `zip()`
- Set
 - unordered collection with no duplicate items
 - `set()` and modification
 - `len()`, `sorted()`, `update()`
 - `add()`, `discard()`
 - `frozenset()`: **immutable**

Set methods

```
>>> set1 = {"a", "b", "e", "f", "g"}
>>> set2 = {"a", "e", "c", "d"}
```

```
>>> set2.add("h")
>>> set2
{'h', 'a', 'c', 'd', 'e'}
```

Add an element to a set.

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

Update a set with the union of itself and others.

Set methods



```
>>> set1 = {"a", "b", "e", "f", "g"}
```

```
>>> set1.discard("a")
```

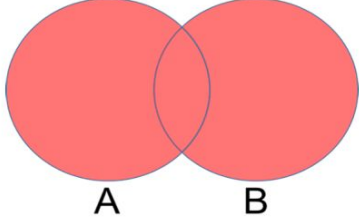
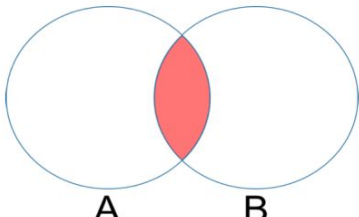
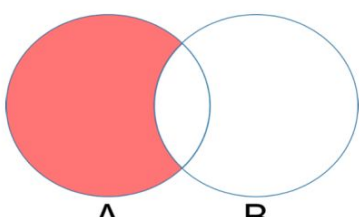
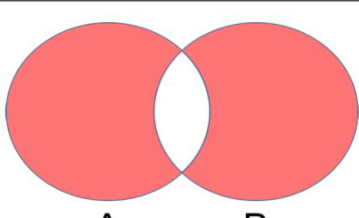
```
>>> set1  
{'g', 'e', 'b', 'f'}
```

*Remove an element from a set if it is a member.
If the element is not a member, do nothing.*

```
>>> set1.clear()
```

```
>>> set1  
set()
```

Remove all elements from this set.

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$, is the set of all values that are a member of A , or B , or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B .
Difference		$A \setminus B$, is the set of all values of A that are not members of B
Symmetric Difference		$A \triangle B$, is the set of all values which are in one of the sets, but not both.

- 基本数据类型
- 控制流
- 函数
- 输入输出

- Conditionals
 - `if` statement
 - `if...else` decision control
 - `if...elif...else` decision control
 - Nested `if` statement
- Loops
 - The `while` loop
 - The `for` loop
 - List comprehensions using `for` loop
 - The `continue` and `break` statements
- `try...except` statement
- `try...except...finally`

Conditionals and loops 条件、循环



- **Loops** : for performing repetitive tasks
- **Conditions**: what to do if it encounters different situations.

Conditionals 条件语句

- `if` statement
- `if ... else` decision control
- `if ... elif ... else` decision control
- Nested `if` statement

```
if Boolean_Expression_1:  
    if Boolean_Expression_2:  
        statement_1  
    else:  
        statement_2  
else:  
    statement_3
```

```
if Boolean_Expression_1:  
    statement_1  
elif Boolean_Expression_2:  
    statement_2  
elif Boolean_Expression_3:  
    statement_3  
:  
:  
else:  
    statement_last
```

```
x = 41
```

```
if x > 10:  
    print("Above ten,")  
    if x > 20:  
        print("and also above 20!")  
    else:  
        print("but not above 20.")
```

Loops 循环语句



- In computer programming, a loop is a statement or a block of statements that is executed repeatedly.
- Python has two kinds of loops
 - a **while** loop
 - a **for** loop

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

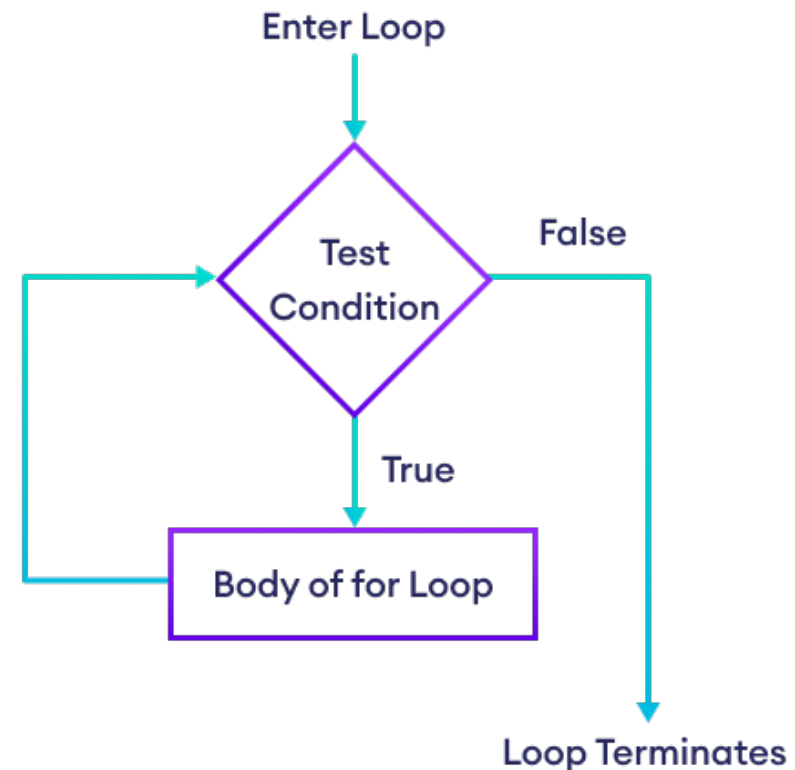
```
In [1]: i = 0  
...: while i < 10:  
...:     print(f"The current number is {i}")  
...:     i += 1  
...:
```

The current number is 0

The current number is 1

.....

The current number is 9



The while loop

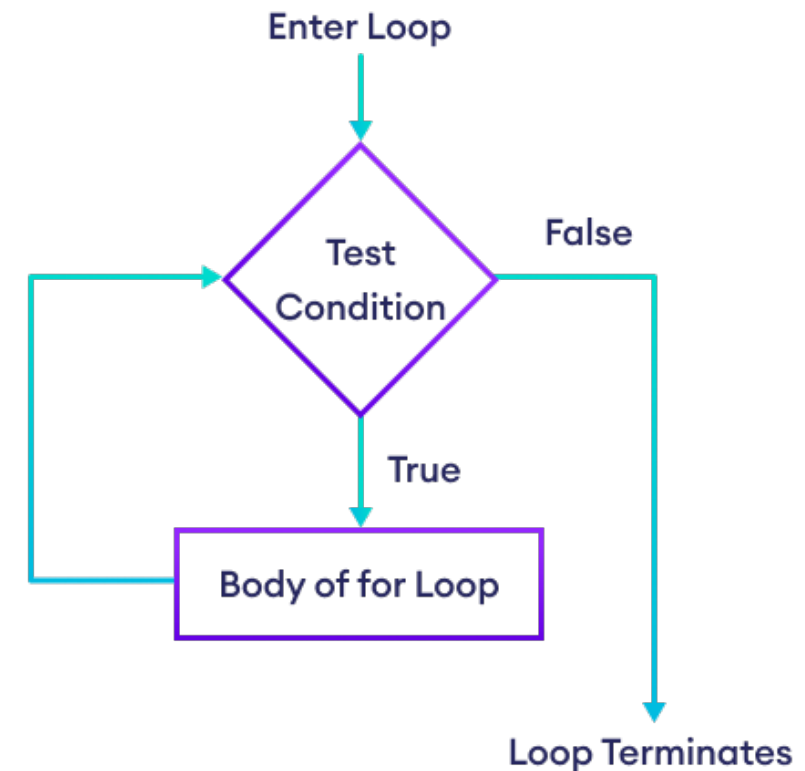
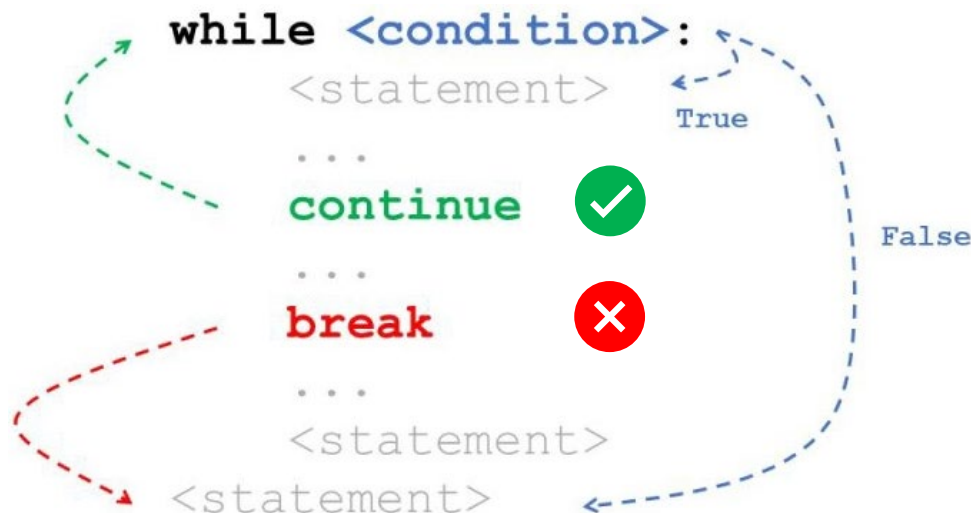


- In computer programming, a loop is a statement or a block of statements that is executed repeatedly.
- Python has two kinds of loops

- a **while** loop

- i: 指针; 条件循环

- a **for** loop



- Whenever the `break` statement is encountered, the execution control **immediately jumps to the first instruction following the loop** 立即跳出循环
- To pass control **to the next iteration** without exiting the loop, use the `continue` statement 立即进行下一次迭代

The for loop

- In computer programming, a loop is a statement or a block of statements that is executed repeatedly.
- Python has two kinds of loops
 - a **while** loop
 - a **for** loop
 - 遍历元素
 - 指定循环次数

```
In [2]: for dogname in ["Molly", "Max", "Buster", "Lucy"]:  
...:     print(dogname+ " Arf, arf!")  
...:     print("All done.")
```

Molly Arf, arf!

Max Arf, arf!

Buster Arf, arf!

Lucy Arf, arf!

All done.

```
In [3]: s = 0  
...: for i in range(1, 100, 2):  
...:     print(i, end=' ')  
...:     s = s+i  
...: print('\n{}'.format(s))  
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47  
49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91  
93 95 97 99  
2500
```

List comprehensions 列表推导式



- List comprehensions are a special feature of core Python for processing and constructing lists. We introduce them here because they use a looping process

```
>>> A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> A
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> diag = []
>>> for i in [0, 1, 2]:
...     diag.append(A[i][i])
>>> diag
[1, 5, 9]
```

- List comprehensions provide a simpler, cleaner, and faster way to build a list of the diagonal elements of A

```
>>> diagLC = [A[i][i] for i in [0, 1, 2]]
>>> diagLC
[1, 5, 9]
```

- Notice here how y serves as a dummy variable accessing the various elements of the list diagLC

```
>>> [y*y for y in diagLC]
[1, 25, 81]
```

List comprehensions using **for** loop



- Extracting a row from a 2-dimensional array such as *A* is quite easy. For example the second row is obtained quite simply in the following fashion:

```
>>> A[1]
[4, 5, 6]
```

- Obtaining a column is not as simple, but a list comprehension makes it quite straightforward:

```
>>> c1 = [a[1] for a in A]
>>> c1
[2, 5, 8]
```

- Another, slightly less elegant way to accomplish the same thing is

```
>>> [A[i][1] for i in range(3)]
[2, 5, 8]
```

- extract all the elements of a list that are divisible by three

```
>>> y = [-5, -3, 1, 7, 4, 23, 27, -9, 11, 41]
>>> [x for x in y if x%3==0]
[-3, 27, -9]
```

try...except statement



- Exceptions: errors detected during execution

```
>>> 10 * (1/0)
```

```
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

```
>>> '2' + 2
```

```
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: Can't convert 'int' object to str implicitly
```

```
In [1]: try:  
...:     print(5/0)  
...: except ZeroDivisionError:  
...:     print("You can't divide by zero!")  
...:  
You can't divide by zero!
```

try:



Run this code

except:



Execute this code when
there is an exception

try...except...finally



- Handling exceptions
- 异常处理确保程序的流程在发生异常时不会中断

```
try:
```

```
    statement_1
```

```
except Exception_Name_1:
```

```
    statement_2
```

```
except Exception_Name_2:
```

```
    statement_3
```

```
    .
```

```
    .
```

```
    .
```

```
else:
```

```
    statement_4
```

```
finally:
```

```
    statement_5
```

- 对可能受异常影响的代码进行分区
- 关联的except块用于处理在try块中抛出的任何结果异常
- 对于每个被抛出的异常，只执行其中一个 except 语句块
- 不论是否发生异常，finally 语句块在离开try 语句时一定会被执行

finally 关键字用于创建 **finally** 块，该块执行在该块中编写的所有语句，而不关心是否引发异常。

Assignment 3-A



Number of Quickest Way Home!

Input

First line: $x y$, which represents your home's location, separated by space.

Second line: number of constructing crossings, n .

Following n lines: $x_n y_n$, which is the location of n th unique constructing crossing, separated by space.

Output

One integer, showing the number of quickest way home.

If SUSTech is your home. output 1.

Unfortunately, 5 crossings are under construction. (This is similar to test case 3, which has 15 points.)

Input #3

Copy

```
10 10
5
1 3
3 5
6 8
8 6
5 5
```

Output #3

Copy

```
37242
```

Assignment 3-C



Sudoku Detection

- Each row must contain the digits 1-9 without repetition.
- Each column must contain the digits 1-9 without repetition.
- Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Input & Output

The path name will be provided in the form of an input text

```
fileName = input()
```

If the Sudoku is valid, then print one line True, otherwise False.

Samplpe type

1.csv:

```
1 5,3,.,.,7,.,.,.,.  
2 6,.,.,1,9,5,.,.,.  
3 .,9,8,.,.,.,6,.  
4 8,.,.,6,.,.,.,3  
5 4,.,.,8,.,3,.,.,1  
6 7,.,.,2,.,.,.,6  
7 .,6,.,.,.,2,8,.  
8 .,.,.,4,1,9,.,.,5  
9 .,.,.,8,.,.,7,8
```

Input #1

Copy

1.csv

Output #1

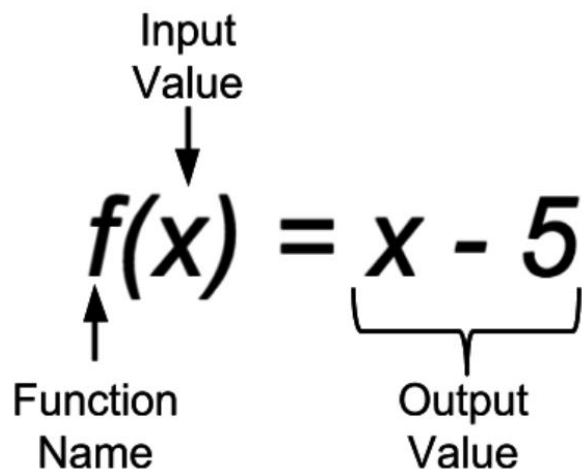
Copy

False

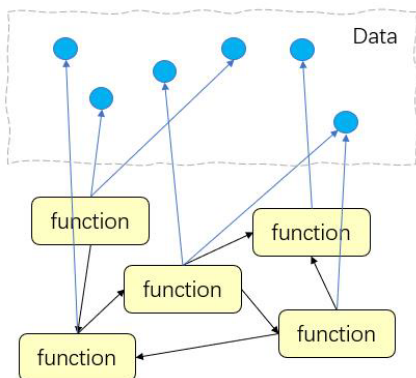
- 基本数据类型
- 控制流
- 函数
- 输入输出

- 基本数据类型
- 控制流
- 函数
- 输入输出
 - 形式参数和实际参数 parameters and arguments
 - 全局变量和局部变量 global and local variables
 - 传入可变和不可变对象 mutable and immutable
 - 默认参数 default parameters
 - 位置参数和关键词参数 positional and keyword args

Programming Paradigms 编程范式



Procedural Design



High coupling. Reduced information hiding.
Hard to make changes and to scale.

Organize data and logics in **objects** (fields and methods)
Object-oriented Design



Traffic Control System

Data
function1
function2
.....



Data
function1
function2
.....



Data
function1
function2
.....



High cohesion. Good information hiding.
Easier to maintain and extend.

Programming Paradigms

Object Oriented Programming

Procedural Programming

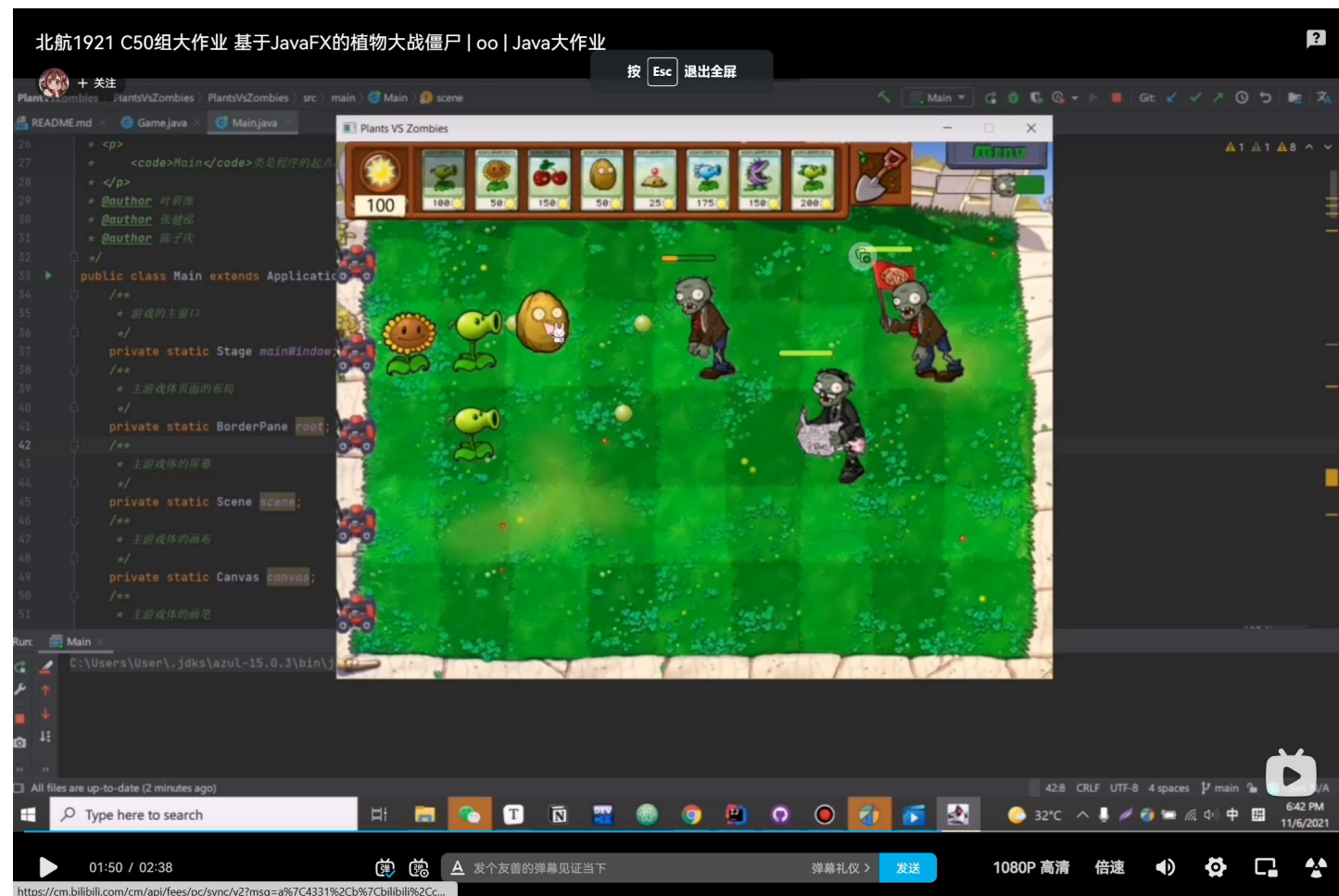
Functional Programming

Organize code as sequential **procedures**
(may change same data)

Organize code as **functions**
(behaviors are separated from the data, which won't be changed)

编程范式好比武功门派，博大精深且自成体系。
——摘自《冒号课堂：编程范式与OOP思想》

- ▶ **Class**—a blueprint
- ▶ **Method**—designed to perform
- ▶ **Object**—the plants we plant
- ▶ **Method call**—perform the task
- ▶ **Instance variable**—to specify the attributes





- 函数式编程的优势:

- 函数的操作灵活

- 函数可作参数传入
 - 函数可对变量赋值
 - 函数可被返回

- 无副作用

- 不可变性

```
function me() {  
  return '👤';  
}  
  
greet(me);
```

```
const greet = function () {  
  console.log('👋');  
}  
  
// The greet variable is now a f  
greet();
```

```
// #3 Return as values from other functions  
function Promises() {  
  return new Promise((resolve, reject) => {  
    resolve('空');  
  }));  
}
```



- 函数式编程的优势:

- 函数的操作灵活

- 无副作用

- 对于相同的输入，函数永远会给出相同的输出

- 不可变性

- 不可变类型的变量一旦被定义，其值不会被改变

- 函数可以帮助避免赋值语句和循环语句

`x = x + 10`

Non-functional style

```
int plusTen(int x)
{
    return x+10;
}
```

Functional style



- 函数式编程的优势：
 - 函数的操作灵活
 - 无副作用
 - 对于相同的输入，函数永远会给出相同的输出
 - 不可变性
 - 不可变类型的变量一旦被定义，其值不会被改变
 - 函数可以帮助避免赋值语句和循环语句

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$$

```
def factorial(n):  
    fact = 1  
    while n >= 1:  
        fact = fact * n  
        n = n - 1  
    return fact
```

Using loops

```
def factorial(n):  
    if n <= 0: return 1  
    return n * factorial(n-1)
```

Using recursive functions (which
invoke themselves)
Or higher order functions (e.g., map)

- Modules in Python are reusable libraries of code having a `.py` extension, which implements a group of methods and statements
- To use a module in your program, import the module using the `import` statement:

```
>>> import math
>>> math.ceil(5.4)
6
>>> math.sqrt(4)
2.0
>>> math.cos(1)
0.5403023058681398
>>> math.factorial(6)
720
>>> math.pow(2, 3)
8.0
```

Function definition

- 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** (形式参数) 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

Function definition

- The syntax for function call or calling function is:

```
function_name(arg_1, arg_2,...,arg_n)
```

- **Arguments (实际参数)** are the actual value that is passed into the **calling function**. There must be a one-to-one correspondence between the parameters in the function definition and the actual arguments of the calling 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

Function definition



- 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__"`:

```
def function_with_no_argument():  
    print("This is a function with NO Argument")
```

```
def function_with_one_argument(message):  
    print(f"This is a function with {message}")
```

```
def main():  
    function_with_no_argument()  
    function_with_one_argument("One Argument")
```

```
if __name__ == "__main__":  
    main()
```

The return statement and void function



- 函数执行任务，并将得到的结果返回给调用函数。返回值的过程可以用函数定义中的可选return语句来实现

```
return [expression_list]
```

- return语句**终止函数定义**，并把一个值返回给调用函数
- Functions without a **return** statement like this are called **void** functions, and they return **None** 无返回函数的返回值为None
- If you want to return a value using the **return** statement from the function definition, then you have to assign the result of the function to a variable
- A function can return only **a single value**, but that value can be **a list or tuple**

Scope and lifetime of variables



- Python programs have two scopes: **global** and **local**
- **Global variable** (全局变量) is accessible and modifiable throughout the program. A variable that is defined **inside a function** definition is a **local variable** (局部变量)
- 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 局部变量伴随函数执行
- **Global variables** are accessible from inside a function, as long as you have not defined a local variable with the same name 全局变量可在函数内调用
- A **local variable** can have the same name as a global variable, but they are totally different so changing the value of the local variable has no effect on the global variable. Only the **local variable** has meaning **inside the function** in which it is defined

```
>>> a = 1
>>> def test_a(x):
    a = 2
    a += x
    return a, x
```

```
>>> local_a, global_a = test_a(a)
>>> print(local_a, global_a)
3 1
```



Scope and lifetime of variables

```
variable = 5
def outer_function():
    variable = 60
    def inner_function():
        variable = 100
        print(f"Local variable of value {variable}")
    inner_function()
    print(f"Local variable of value {variable}")
outer_function()
print(f"Global variable of value {variable}")
```

- 全局变量可能会影响函数，函数内的局部变量不会影响全局变量
- 不建议从函数定义内部访问全局变量
- 建议将全局变量作为参数传入函数

Passing mutable and immutable objects



- 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**
- Changes to **immutable arguments** of a function within the function do not affect their values in the calling program 调用函数时，不可变类型的全局变量不会被改变
- Lists and arrays are **mutable**; those elements that are changed inside the function are also **changed** in the calling function 调用函数时，可变类型的全局变量会被改变
- Changes to mutable arguments of a function within the function are reflected in the values of the same list and array elements in the calling function

Passing mutable and immutable objects



String

```
def test(s):  
    s = "Hello again"  
    return s
```

```
s = "Hello"  
s1 = test(s)  
print(f"s = {s}")  
print(f"s1 = {s1}")  
print(f"s = {s}")
```

```
s = Hello  
s1 = Hello again  
s = Hello
```

Number

```
def test(n):  
    n = 100  
    return n
```

```
n = 30  
n1 = test(n)  
print(f"n = {n}")  
print(f"n1 = {n1}")  
print(f"n = {n}")
```

```
n = 30  
n1 = 100  
n = 30
```

Tuple

```
def test(t):  
    t = (1, 2)  
    return t
```

```
t = (3, 4)  
t1 = test(t)  
print(f"t = {t}")  
print(f"t1 = {t1}")  
print(f"t = {t}")
```

```
t = (3, 4)  
t1 = (1, 2)  
t = (3, 4)
```

Passing mutable and immutable objects



List

```
def test(l):  
    l[-1] = "end"  
    return l  
  
l = [1, 2]  
print(f"l = {l}")  
l1 = test(l)  
print(f"l1 = {l1}")  
print(f"l = {l}")
```

```
l = [1, 2]  
l1 = [1, 'end']  
l = [1, 'end']
```

Numpy array

```
import numpy as np  
def test(a):  
    a[0] = 100  
    return a  
  
a = np.zeros(2)  
print(f"a = {a}")  
a1 = test(a)  
print(f"a1 = {a1}")  
print(f"a = {a}")
```

```
a = [0.  0.]  
a1 = [100.  0.]  
a = [100.  0.]
```

Default parameters 默认参数

- Usually, the default parameters are defined at the end of the parameter list:

```
def work_area(prompt, domain="Bioinformatics"):  
    print(f"{prompt} {domain}")  
work_area("Sam works in")  
work_area("Alice has interest in", "Genomics")
```

- Output:

```
Sam works in Bioinformatics  
Alice has interest in Genomics
```

Positional and keyword arguments



- Generally, whenever you call a function with some values as its arguments, these values get assigned to the parameters in the function definition according to their position
- You can also explicitly specify the **keyword argument** name along with its value in the form **kwarg = value**. The normal arguments without keywords are called **positional arguments**
- In the calling function, keyword arguments must follow positional arguments
- All the keyword arguments passed must match one of the parameters in the function definition and their order is not important
- No parameter in the function definition may receive a value more than once

Positional and keyword arguments



```
def area_rectangle(a, b = 6):  
    area = a * b  
    print(f'Side a is {a}, side b is {b}')
```

```
print(f'The area is {area}')
```

```
>>> area_rectangle(4)  
Side a is 4, side b is 6  
The area is 24
```

```
>>> area_rectangle(a = 4)  
Side a is 4, side b is 6  
The area is 24
```

```
>>> area_rectangle(4, 5)  
Side a is 4, side b is 5  
The area is 20
```

```
>>> area_rectangle(5, 4)  
Side a is 5, side b is 4  
The area is 20
```

```
>>> area_rectangle(a = 4, b = 5)  
Side a is 4, side b is 5  
The area is 20
```

```
>>> area_rectangle(b = 5, a = 4)  
Side a is 4, side b is 5  
The area is 20
```

Positional and keyword arguments



```
def area_rectangle(a, b = 6):  
    area = a * b  
    print(f'Side a is {a}, side b is {b}')  
    print(f'The area is {area}')
```

Invalid function calls:

<pre>>>> area_rectangle()</pre>	<pre>TypeError: area_rectangle() missing 1 required positional argument: 'a'</pre>
<pre>>>> area_rectangle(a = 5, 4)</pre>	<pre>SyntaxError: positional argument follows keyword argument</pre>
<pre>>>> area_rectangle(5, a = 5)</pre>	<pre>TypeError: area_rectangle() got multiple values for argument 'a'</pre>
<pre>>>> area_rectangle(5, c = 4)</pre>	<pre>TypeError: area_rectangle() got an unexpected keyword argument 'c'</pre>



*args and **kwargs

- `*args` and `**kwargs` allow you to pass **a variable number of arguments** to the function definition. It is very useful when user does not know in advance about how many arguments will be passed to the function definition
- `*args` as parameter in function definition allows you to pass **a non-keyworded & variable length tuple** argument list to the function definition
- `**kwargs` as parameter in function definition allows you to pass **keyworded & variable length dictionary** argument list to the function definition

*args and **kwargs

- `*args` must come after all the positional parameters and `**kwargs` must come right at the end
- The single asterisk (*) and double asterisk (**) are the important elements here and the words `args` and `kwargs` are used only by convention. The Python does not enforce those words and the user is free to choose any words of their choice



*args and **kwargs

```
def fruit_shop(*args, **kwargs):  
    print(args, kwargs)  
    fruits = ", ".join(args)  
    print(f"Do you have: {fruits}?")  
    print(f"Yes, the price is:")  
    for kw in kwargs:  
        print(f"{kw}: ${kwargs[kw]}/each")  
  
>>> fruit_shop("banana", "orange", banana = 3, orange = 7)  
( 'banana', 'orange' ) { 'banana': 3, 'orange': 7 }  
Do you have: banana, orange?  
Yes, the price is:  
banana: $3/each  
orange: $7/each
```

Command line arguments



example.py

```
import sys
def main():
    for arg in sys.argv:
        print(arg)
if __name__ == "__main__":
    main()
```

In your terminal:

```
$ python example.py first_arg 100 10.0
main.py
first_arg
100
10.0
```

- 基本数据类型
- 控制流
- 函数
- 输入输出

- **Input**

- Keyboard input

```
strname = input("prompt to user")
```

```
Strname = eval(strname)
```

- Input data file

- **Output**

- Screen output
 - Output data file

- **Files**

- Text files
 - Binary files
 - Headers: identifies the file's contents
 - Path: absolute path, relative path

File path

- A file path can be **absolute path** or **relative path**
- A path is an absolute path if it points to the file location, which always contains the root and the complete directory list
- The root directory contains all other directories in the drive. The main root directory of the Windows system is **C:** and the root directory in Linux system is **/**(forward slash)
- A path is a relative path if it contains “double-dots” or “single-dot” :
 - "**..\langur.txt**" specifies a file named "**langur.txt**" located in the parent of the current directory
 - "**.\bison.txt**" specifies a file named "**bison.txt**" located in a current directory
 - "**..\..\langur.txt**" specifies a file that is two directories above the current directory

Creating files

- All files must be opened first before they can be read from or written to using the Python's built-in `open()` function:

```
file_handler = open(filename, mode)
```

- `filename` is a **string** containing the file path to be opened
- `mode` (optional) is a **string** describing the way the file is used

Mode	Description
"r"	open the file in read only mode (default)
"w"	open the file for writing. if exists, it'll be overwritten; else, creates a new file
"a"	open the file for appending data at the end of the file automatically
"r+"	open the file for both reading and writing
"w+"	open the file for reading and writing. if exists, it'll be overwritten; else, creates a new file
"a+"	open the file for reading and appending. if exists, the data will be appended; else, creates a new file
"rb"	open the binary file in read-only mode
"wb"	open the file for writing the data in binary format

Creating files

- The `open()` method returns a file handler object that can be used to read or modify the file:

```
>>> file_handler = open("moon.txt", "r")
>>> file_handler = open("C:\\langur.txt", "r")
>>> file_handler = open("C:\\prog\\example.txt", "r")
>>> file_handler = open("../\\bison.txt", "r")
```
- The `close()` method close the file once the processing is completed:

```
>>> file_handler = open("moon.txt", "r")
>>> file_handler.close()
```
- If the file is not closed explicitly, Python's garbage collector will eventually destroy the object and close the opened file, but the file may have stayed open for a while

with statement



- The **with** statement automatically closes the file after executing its block of statements: **with的优势：可以自动关闭文件资源**

```
with open (file, mode) as file_handler:  
    Statement_1  
    Statement_2
```

- Example:

```
def read_file():  
    print("Printing each line in the text file")  
    with open("file_test.txt") as file_handler:  
        for each_line in file_handler:  
            print(each_line, end="")  
  
def main():  
    read_file()  
  
if __name__ == "__main__":  
    main()
```

File object attributes



- The file handler has various attributes:

```
>>> fh = open("computer.txt", "w")
>>> fh.name
'computer.txt'
>>> fh.closed
False
>>> fh.mode
'w'
```

Attribute	Description
<code>fh.closed</code>	return a Boolean <code>True</code> if the file is closed or <code>False</code> otherwise
<code>fh.mode</code>	return the access mode with which the file was opened
<code>fh.name</code>	return the name of the file

File methods to read and write data

- List of methods associated with the file object:

Method	Description
<code>fh.read([size])</code>	read the contents of a file up to a size and return as a string. The argument <code>size</code> is optional, and, if not specified, the entire contents of the file will be read and returned
<code>fh.readline()</code>	read a single line in file
<code>fh.readlines()</code>	read all the lines of a file as list items
<code>fh.write()</code>	write the contents of the string to the file, returning the number of characters written
<code>fh.writelines()</code>	write a sequence of strings to the file
<code>fh.tell()</code>	return an integer giving the file handler's current position within the file, measured in bytes from the beginning of the file
<code>fh.seek(offset, from_what)</code>	change the file handler's position. The position is computed from adding <code>offset</code> to a reference point. The reference point is selected by the <code>from_what</code> argument. A <code>from_what</code> value of 0 measures from the beginning of the file (default), 1 uses the current file position, and 2 uses the end of the file as the reference point

The `pickle` module

- The `read()` method only returns strings. However, things get a lot more complicated when you want to save more complex data types like lists, dictionaries, or class instances
- The `pickle` module can take almost any Python object and convert it to a string – [pickling](#):

```
pickle.dump(obj, file_handler)
```
- Reconstructing the object from the string representation is called [unpickling](#):

```
obj = pickle.load(file_handler)
```

Read and write CSV files

- CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases
- Columns are separated with commas, and rows are separated by line breaks or the invisible “\n” character. However, the last value is not followed by a comma

CSV file "contact.csv":

```
name, email, mobile
john, john@gmail.com, 555-0134
will, will@yahoo.com, 888-3456
jane, jane@outlook.com, 777-0189
```

contact.csv			
1	name, email, mobile		
2	john, john@gmail.com, 555-0134		
3	will, will@yahoo.com, 888-3456		
4	jane, jane@outlook.com, 777-0189		

File Home Insert Page Layout Formulas Data				
Clipboard		Font		
A1		name		
	A	B	C	D
1	name	email	mobile	
2	john	john@gmail.com	555-0134	
3	will	will@yahoo.com	888-3456	
4	jane	jane@outlook.com	777-0189	
5				

Spaces after “,” are a part of the field

- *Human-readable format and easy to edit manually*
- *Simple to generate, parse and handle*
- *It is a standard format and is supported by many applications*

```
['name', ' email', ' mobile']
['john', ' john@gmail.com', ' 555-0134']
['will', ' will@yahoo.com', ' 888-3456']
['jane', ' jane@outlook.com', ' 777-0189']
```

Read and write CSV files

- `csv` module can be used to work with CSV files:
`import csv`
- To read from a CSV file use `csv.reader()` method:
`csv.reader(csvfile)`
- To write to a CSV file, use the `csv.writer()` method:
`csvwriter = csv.writer(csvfile)`
`csvwriter.writerow(row)`
`csvwriter.writerows(rows)`



Thanks!

