Python Grammar

A. Data types

1. single types

(1) integers & floats

booleans: True, False

(2) knowing the data type.

```
number_of_dogs = 1
print(type(number_of_dogs))
```
➡ `<class 'int'>` — *An integer*

```
age_of_my_dog = 1.5
print(type(age_of_my_dog))
```
➡ `<class 'float'>` — *A float*

```
name_of_my_dog = "Toffee"
print(type(name_of_my_dog))
```
➡ `<class 'str'>` — *A string*

(3) data_type conversion

The number is stored as an integer

```
mynumber = int(5)
print(mynumber)
```
➡ `5`

The number is stored as a float

```
mynumber = float(5)
print(mynumber)
```
➡ `5.0`

⚠ A problem & the safer approach

```
age = "21.5"
age = int(float(age))
print(age)
```
➡ `21`

```
age = 21
print("I am " + str(age) + " years old!")
```
➡ `I am 21 years old!`

2. collection

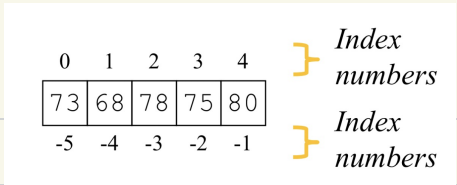
(1) list

a) list is used to store many things together,
the items can be of any type (OK to be different)
eg. ["John", 18, "CSE", 180]
(0, 1, 2, 3)

b) navigating

i) use len(listname) to find how many items in a list.

ii) indicies



c) handling of lists

- Creating a list:
  list_name = [*first_thing*, *second_thing*, …]
- Reading a value from the list:
  list_name[*item_number*]
- Changing a value in the list:
  list_name[*item_number*] = *new_thing*
- Inserting a value into the list:
  list_name`.insert(` *position*, *new_thing*)
- Removing something from the list (once):
  list_name`.remove(` *thing_you_want_to_remove*)

Works for strings as well

To go through a list:

① `for i in range(len(l)):`
`  print(l[i])`

② `for i in l:`
`  print(i)`

e.g. 把new_thing放到i，后面往后挪一位

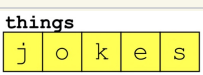
- Adding something new at the end:
  list_name`.append(` *thing_you_want_to_append*)
- Sorting the list: → from small to large
  list_name`.sort()`
- Reversing the order of the things in the list:
  list_name`.reverse()`
- Counting something in the list:
  list_name`.count(` *thing_you_want_to_count*)
- Searching for something in the list:
  list_name`.index(` *thing_you_are_searching_for*)
- Adding another list at the end of the list:
  list_name`.extend(` *another_list*)

Works for strings as well

d) 2-D structures

- 1D example:

```
things = ["j","o","k","e","s"]
letter_o = things[1] # the "o" in second col
```



- 2D example:

```
things =
 [["h","a","r","r","y"],
  ["l","i","k","e","s"],
  ["j","o","k","e","s"]]
letter_o = things[2][1]
# the "o" in the third row second column
```

Be careful! The general idea is `[row][col]`, not `[col][row]`!

(2) tuple

a) A similar idea to lists, cannot change once created!

b) creation: use `()` instead of `[]`

c) handling

len(), count(), index() → work

remove(), append(), sort(), reverse(), extend() → don't work

d) string.

- a string is a tuple of letters
- we can concatenate strings to create a new string:

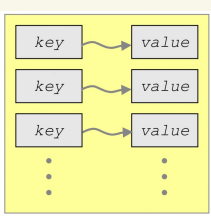
  two_words = "pretty" + "umbrellas" (exactly together)

  "fun" * 3 = "funfunfun"
- as long as not changed, handling of lists works

(3) dictionary

a) about dictionary

- The left side is called the *key*
- The right side is called the *value*
- We say that the key *is mapped to* the value



→ any thing can be used for the keys or values (eg. texts, ints, floats, etc.)

BUT: Cannot use a list as key.

```
animals = {
 "ant":"a small insect with 6 legs",
 "dog":"an animal with 4 legs that goes woof",
 "cat":"an animal with 4 legs that goes miaow"
}
result=animals["ant"]
print(result)
```

The dictionary doesn't have to be in alphabetical order!

b) retrieve

david_data = heads["David"] // width = david[2]

c) handling a dictionary

· creating

```
heads = {"David":  (589, 106, 48, 63),
         "Gibson": (474, 102, 44, 58),
         "Paul": (522, 162, 55, 68)
}
```

x position of top left corner — *y position of top left corner* — *width* — *height*

· adding

```
>>>print(heads)
{'David': (589, 106, 48, 63), 'Gibson':
(474, 102, 44, 58), 'Paul': (522, 162,
55, 68)}
>>>heads["Sean"] = (628, 146, 46, 58)
>>>print(heads)
{'David': (589, 106, 48, 63), 'Gibson':
(474, 102, 44, 58), 'Paul': (522, 162,
55, 68), 'Sean': (628, 146, 46, 58)}
```

· delete

```
>>>print(heads)
{'David': (589, 106, 48, 63), 'Gibson':
(474, 102, 44, 58), 'Paul': (522, 162,
55, 68)}
>>>del heads["Paul"]
>>>print(heads)
{'David': (589, 106, 48, 63), 'Gibson':
(474, 102, 44, 58)}
```

'Paul' data is gone!

· change value

```
>>>print(heads)
{'David': (589, 106, 48, 63), 'Gibson':
(474, 102, 44, 58), 'Paul': (522, 162,
55, 68)}
>>>heads["David"]=(588, 104, 48, 57)
>>>print(heads)
{'David': (588, 104, 48, 57), 'Gibson':
(474, 102, 44, 58), 'Paul': (522, 162,
55, 68)}
```

Changed

d) going through a dictionary

```
for key in heads.keys():
    print(key)
```

```
for value in heads.values():
    print(value)
```

```
for key, value in heads.items():
    print(key, value)
```

* Slicing Notation

a) name_of_list_or_tuple[Start:End:Step]

Let's assume we have a list x, which looks like this:

```
x = [1, 2, 3, 4, 5]
    0  1  2  3  4
```

Here are some examples of slicing:

- `x[0:3]` returns `[1, 2, 3]`
- `x[0:5:2]` returns `[1, 3, 5]`
- `x[3:]` returns `[4, 5]`
- `x[:3]` returns `[1, 2, 3]`
- `x[4:0:-1]` returns `[5, 4, 3, 2]`

FYI: negative indicies have a special meaning in python, so x[4:-1:-1] won't work (-> return [])

Where is the first item?

- Instead of using `-1` as the end number you can return the reversed list of x like this:

`x[4::-1]`

or simply omit both the start and end numbers:

`x[::-1]`

slice4.py - //VDIRIVE/MYHOME/rossiter/Documents/slice4.py

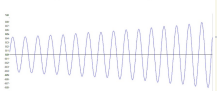
File Edit Format Run Options Window Help

```
x = [1, 2, 3, 4, 5]
print(x[4::-1])
print(x[::-1])
```

```
[5, 4, 3, 2, 1]
[5, 4, 3, 2, 1]
```

b) application: Digital Audio

- A sound file consists of a sequence of values, called audio *samples* (a sample is simply a number)
- These audio samples can be positive or negative
- The sequence of values forms the shape of the sound wave, which represents the sound



Some audio samples, shown in audio editing software — *Time*

Accessing Precise Sections

- Digital audio uses a fixed number of samples for each second
- In the COMP1021 WAV files, 44100 samples are used for every second of audio

```
# Access the first second of the audio
samples[:44100]
# Access the third second of the audio
samples[44100*2:44100*3]
# Access the third second of the audio backwards
samples[44100*3:44100*2:-1]
```

B. Basic Grammar and Functions

1. Calculating

1) operators

```
response = input("Are you alive? (yes/no)")
response = response == "yes"
```

来写不可省略

Arithmetic Operators

- Basic operators:
  `+ - / * %`
- 'Advanced' operators:
  - `**` means 'to the power of'
  - `//` means 'do division, return the integer result'
  - `-x` means the same as '`-1 * x`'

```
2**3
8
3**2
9
3//3
1
4//3
1
5//3
1
6//3
2
7//3
2
8//3
2
x=10
-x
-10
```

Operators for Lists, Tuples and Strings

- These operators are used by lists, tuples and strings:
  - `x + y` concatenates (=put together) two lists, tuples or strings
  - `x * n` concatenates `n` copies of `x`
  - `a in x` returns `True` if `a` is in collection `x` and `False` otherwise
  - `a not in x` returns `False` if `a` is in collection `x` and `True` otherwise

COMP1021 More on Operators Page 12

Comparison Operators

Reminder

- For comparing two values:
  - `a < b` returns `True` if `a` is less than `b`
  - `a <= b` returns `True` if `a` is less than or equal to `b`
  - `a > b` returns `True` if `a` is greater than `b`
  - `a >= b` returns `True` if `a` is greater than or equal to `b`
  - `a == b` returns `True` if `a` is equal to `b`
  - `a != b` returns `True` if `a` is not equal to `b`
- All of them return `False` otherwise

Using 'in' with Strings

- Using the `in` operator you can test for a string inside another string, like this:

```
if "shark" in "baby shark dance":
    print("yes")
```

```
yes
```

Logical Operators

Reminder

- Logical operators work with Boolean values, i.e. `True` or `False`
  - `a and b` if both condition *a* and condition *b* are True, the result is True; otherwise, it's False
  - `a or b` if either condition *a* or condition *b* is True, the result is True; otherwise, it's False
  - `not a` if *a* is True, then the result is False; if *a* is False, then the result is True

Using Other Things as True/False

- Any number other than 0 means `True`
  - 0 means `False`
- An empty list `[]`, tuple `()` or string `""` means `False`
  - Non-empty list/ tuple/ string means `True`

2) precedence

Increasing precedence

- *Highest precedence* -
  - `( )`
  - `**`
  - `-x, +x`
  - `*, /, %, //`
  - `+, -`
  - `<, >, <=, >=, !=, ==`
  - `in, not in`
  - logical `not`
  - logical `and`
  - logical `or`
- *Lowest precedence* -

So if you use brackets `()` they override everything

2. input and output

1) x = input("Your question here?")
   - the inputed content is always a str.
   - need to convert it necessary: x = int(x)

2) print()

   a> print("...", end="x"), print x in the end, (default="\n")

   b> print("a" + "b"), print things after normal "+" ! only for the same data type to do so!

   c> print("a", "b"), a space in the middle.

3. conditional statement.

1) statements: if, if-else, if-elif, nested if.

```
if a>b:
    ......
else:
    ......
```

```
if a>b:
    ......
elif c>d:
    ......
else:
    ......
```

```
if a>b:
    if c>d:
        ...
else:
    if c>d:
```

2) comparison operators: <, <=, >, >=, ==, !=

3) logical operators: ( ) > not > and > or

4. loop statement

1) while statement

```
while ... condition ... :
    ... statement(s) ...
statements after loop
```

```
while True:
    x = input("whats your choice?")
    ......
```

```
while ... condition ... :
    ......
    while ... condition2 ... :
```

2) for statement

```
for i in <list>:
    ... statement(s) ...
```

```
for _ in range(5):
    ... statement(s)
```
<= if the "_" is not referred to anywhere inside the loop

eg. using loop statements and % to handle repeating patterns

3) break & continue -> apply for the loop they're in

5. functions

1) how to define

- Here is an example:

In this example, the function is expected to receive a value, stored in a variable called 'name'

```
def show_response(name):
    if name == "Dave":
        print("What a good name!")
    else:
        print("How are you?")
```

=> must be defined before used.

2) global & local variables

You need to be careful when you change a local variable:

```
def magic_trick(money):
    if money < 1000:
        money = money + 500

money = int(input("How much do you have? "))
magic_trick(money)
print("You have $" + str(money) + " now!")
```

The local variable is changed in this line of code

The global variable `money` *is not affected by the change inside the function*

```
How much do you have? 500
You have $500 now!
```

- If you want a global variable to be changed by a function you need to tell Python using the `global` command, for example:

```
def magic_trick():
    global money

    if money < 1000:
        money = money + 500

money = int(input("How much do you have? "))
magic_trick()
print("You have $" + str(money) + " now!")
```

We tell Python that when we refer to `money` *in the function, it means the global variable* `money`

This line changes the value of the global variable

⇒ different approaches

3) returning values from a function

Returning Multiple Things

- We can return more than one thing
- E.g. the following function returns two values:

```
def get_info(current_year, year_of_birth):
    chinese_zodiac = [
        "Rat", "Ox", "Tiger", "Rabbit",
        "Dragon", "Snake", "Horse", "Sheep",
        "Monkey", "Rooster", "Dog", "Pig"
    ]

    age = current_year - year_of_birth
    animal = chinese_zodiac[  \
        (year_of_birth - 1960) % 12 ]

    return age, animal
```

Two values are returned in this example

Getting Multiple Results

- To get the two results from the function we use two variables, like this:

```
year = int(input("Hi, what is the current year? "))
birthyear = int(input("When is your year of birth? "))

yourage, youranimal = get_info(year, birthyear)

print("You are", yourage)
print("Your animal is", youranimal)
```

```
Hi, what is the current year? 2021
When is your year of birth? 2001
You are 20
Your animal is Snake
```


Stopping a Function Using Return

The complete program:

```
def donate(money):
    if money <= 0:
        return

    print("Thank you! You are so generous!")

donation = int(input("How much do you donate? "))
donate(donation)

print("Finished!")
```

```
How much do you donate? -5000
Finished!
```

```
How much do you donate? 100
Thank you! You are so generous!
Finished!
```

If the `return` *command is executed then the function immediately finishes, and Python continues with any code under the place where the function was executed*

6. range problem.

- Here are more examples of using the step value:
  - `range(0, 10, 3)` returns 0, 3, 6 and 9
  - `range(-1,-10,-2)` returns -1, -3, -5, -7 and -9
- Here are some unusual examples of using `range()`:
  - `range(10, 1)`
    returns nothing because the default step value is 1
  - `range(-10, 1,-1)`
    returns nothing because the step value is -1
  - `range(0, 10, 0)`
    results in an error because the step value must not be zero

number of rounds:

$$\left\lceil \frac{y-x}{d} \right\rceil$$

7. random problem

```
import random
import string

# 随机整数：
print random.randint(1,50)

# 随机选取0到100间的偶数：
print random.randrange(0, 101, 2)

# 随机浮点数：
print random.random()
print random.uniform(1, 10)

# 随机字符：
print random.choice('abcdefghijklmnopqrstuvwxyz!@#$%^&*()')

# 多个字符中生成指定数量的随机字符：
print random.sample('zyxwvutsrqponmlkjihgfedcba',5)

# 从a-zA-Z0-9生成指定数量的随机字符：
ran_str = ''.join(random.sample(string.ascii_letters + string.digits, 8))
print ran_str

# 多个字符中选取指定数量的字符组成新字符串：
print
''.join(random.sample(['z','y','x','w','v','u','t','s','r','q','p','o','n','m','l
','k','j','i','h','g','f','e','d','c','b','a'], 5))

# 随机选取字符串：
print random.choice(['剪刀', '石头', '布'])

# 打乱排序
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
print random.shuffle(items)
```

& text file handling

1) special characters

a) '\t' 向上取整至8n位

b) '\n' 换行，会被默认加在print()末尾

- Here we turn off the default behaviour of print, to make the example easier to understand

```
for x in range(5):
    print( "hello" + "\n" * x, end="")
```

```
hellohello
hello

hello

hello
```

* has a higher precedence than + so this part is done first

The first value generated by `range(5)` is zero, so there's no end-of-line character here

Another Example

2) writing the file & reading the file

```
filename=turtle.textinput("Save jigsaw positions", \
    "What is the jigsaw filename you want to create?")
myfile = open(filename, "wt") #Open the file for writing
# Now we go through each turtle in the list of turtles
for thisTurtle in allTurtles:

    # Make a string for one turtle, in the right format
    one_line = str(thisTurtle.xcor()) + "\t" + \
        str(thisTurtle.ycor()) + "\n"

    # Save the string to the file
    myfile.write(one_line)

# Close the file
myfile.close()
```

Use any name to 'point' to the file

Put a tab between the two text

Add the end-of-line character at the end of the line

It's possible to have several files open at the same time, so you need to say which file you are referring to

```
filename=turtle.textinput("Load jigsaw positions", \
    "What is the jigsaw filename you want to load?")
myfile = open(filename, "r") # Open the file for reading
turtleIndex=0
for line in myfile:
        # Handle each line, one by one
        line = line.rstrip()    # Remove the end-of-line

        items = line.split("\t") # Separate the two items

        x=float(items[0])        # Convert x to a float
        y=float(items[1])        # Convert y to a float

        allTurtles[turtleIndex].goto(x, y) # Move turtle
        turtleIndex=turtleIndex+1 # Increase the index,
                                  # for the next turtle
myfile.close()  # We have finished, now close the file
```

You can use any variable name to 'point to' the file, it doesn't have to be the same one used before

& Other things

- use "\" at the end of line ⇒ the code continues on following line
- # for single-line annotation "'...'" """..""" for multi-line annotation
- $x, y = a, b \iff \begin{matrix} x=a \\ y=b \end{matrix}$
- use "control + enter" to continue codes in IDLE