Programming for Business Computing

| Casting, Input/output, and Assignment

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Casting

- We may convert a value from one type to another type.
  - Type conversion is called **casting** (型別轉換).
- To cast a float or a string to an integer, use **`int()`**.

```
f = 52.0
i = int(f)
print(f)
print(i)
print(type(f))
print(type(i))
```

```
s = "52"
i = int(s)
print(s)
print(i)
print(type(s))
print(type(i))
```

- What will happen if we try to cast 52.6 or "52 is great" to an integer?

Casting

- To cast an integer or a string to an float, use **`float()`**.

```
i = 52
f = float(i)
print(i)
print(f)
print(type(i))
print(type(f))
```

```
s = "52"
f = float(s)
print(s)
print(f)
print(type(s))
print(type(f))
```

- Casting an integer to a float creates no error.
- What will happen if we try to cast "52 is great" to a float?

Casting

- To cast an integer or a float to a string, use **`str()`**.

```
i = 52
s = str(i)
print(i)
print(s)
print(type(i))
print(type(s))
print(len(s))
```

```
f = 52.0
s = str(f)
print(f)
print(s)
print(type(f))
print(type(s))
print(len(s))
```

  - **`len()`** returns the **length** (i.e., number of characters) of a string.

More about `input`

- The function **`input`** reads a user input from the keyboard (typically).
- Whatever the user types, **`input`** read it as a string.
  - Sometimes we need to cast the input by ourselves.
- What is the difference between these two programs?

```
num1 = int(input())
num2 = int(input())
print(num1 + num2)
```

```
num1 = input()
num2 = input()
print(num1 + num2)
```

  - Strings are **concatenated** by the string concatenation operator (**+**).

More about `print`

- The function **`print`** prints whatever behind it.
  - Those things are actually converted to strings before being printed.
- As strings can be **concatenated**, we may put multiple pieces of variables/values (sometimes called "tokens") behind a **`print`** to print all of them.
  - To do the separation, use the comma operator (**`,`**).
- As an example:

```
num1 = int(input())
num2 = int(input())
print("the sum is", num1 + num2)
```

  - There are two items in this print operation.
  - The second item **`num1 + num2`** is first **cast to a string**.
  - The two strings are then concatenated to form a string to be printed out.

More about `print`

- Note that there is a **white space** between “s” and the sum.

```
num1 = int(input())
num2 = int(input())
print("the sum is", num1 + num2)
```

  - Python **automatically** insert a white space between two neighboring items.

- Sometimes it is bad:

```
income = int(input())
print("My income is $", income)
```

- How to remove the space between the dollar sign and **`income`**?

More about `print`

- There are many ways in Python to remove the white spaces.
- The easiest way (though may not be the best way) is to **concatenate** those items into a string **manually** (using **+**).

```
income = int(input())
print("My income is $" + str(income))
```

  - We need to first **cast** **`income`** (or any other non-string items) **into a string** by **`str()`** to avoid a run-time error.

More about `print`

- As another example, to print out two input numbers as a vector, we may:

```
num1 = int(input())
num2 = int(input())
print("the vector is (", num1, ",", num2, ")")
```

- To remove the three bad white spaces, we may:

```
num1 = int(input())
num2 = int(input())
print("the vector is (" + str(num1) + ",", str(num2) + ")")
```

or (which one is better?)

```
num1 = int(input())
num2 = int(input())
print("the vector is (" + str(num1) + ", " + str(num2) + ")")
```

Assignment

- When we put a variable at the left of an **assignment operator** (`=`, 指派運算子), we assign the right-hand-side (RHS) value to it.
- Is the following operation valid?

```
a = 10
a = a + 2

print(a)
```

- **`a = a + 2`** does the following:
  - Finding the value at its right: the value of **`a + 2`** is 12.
  - Assigning that value to the variable at its left: **a becomes 12**.
  - It has nothing to do with the mathematical equality $a = a + 2$ (which cannot be satisfied)!

Self assignment

- A statement like `a = a + 2` is a **self-assignment** (自我指派) operation.
  - A variable is modified according to its own value.
- As self assignment is common, there are self-assignment operators:
  - `a += 2` means `a = a + 2`.
  - `a -= 2` means `a = a - 2`.
  - We also have `*=`, `/=`, `//=`, `**=`, `%=`, etc.

```
a = 10
a += 2
print(a)
a -= 2
print(a)
a *= 2
print(a)
a //= 2
print(a)
a /= 2
print(a)
a **= 2
print(a)
a %= 2
print(a)
```

Cascade assignment

- Is the following operation valid?

```
a = b = 10
a = a + 2

print(a)
print(b)
```

- `a = b = 10` assigns 10 to both **`a`** and **`b`**.
  - This is call **cascade assignment** (層遞指派).
- More variables may be assigned the same value in one statement.
  - And of course, they are different variables.

Programming for Business Computing

| Conditionals (1)

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Conditionals

- So far all our programs execute statements line by line.
- In practice, we may **select** what to do (or what to skip) upon some **conditions**.
- To do the selection, we use **conditionals** (條件判斷).
- In Python, we use **`if`**, **`else`**, and **`elif`**.

The first example

- The income tax rate often varies according to the level of income.
  - E.g., 2% for income below $10000 but 8% for the part above $10000.
- How to write a program to calculate the amount of income tax based on an input amount of income?

```
print("Please enter your income:")
income = float(input())

if income <= 10000:
  tax = 0.02 * income
if income > 10000:
  tax = 0.08 * (income - 10000) + 200

print("Tax amount: $" + str(tax))
```

The `if` statement

- We use the **`if`** statement to control the sequence of executions.

```
if condition:
  statements
```

  - If ***condition*** is **true**, do ***statements*** sequentially.
  - Otherwise, skip those ***statements***.
  - The ***statements*** are said to be inside **the `if` block**.

```
print("Please enter your income:")
income = float(input())

if income <= 10000:
  tax = 0.02 * income
if income > 10000:
  tax = 0.08 * (income - 10000) + 200

print("Tax amount: $" + str(tax))
```

The `if` statement

- The **colon** (`:`) is required.

```
a = 0
if a < 1
  print("a < 1")
```

- There can be multiple statements inside an **`if`** block.
- Statements inside an **`if`** block must all have one level of **indention** (縮排).

```
a = 0
if a < 1:
  print("a < 1")

print("great!")
```

- Statements with no indention are considered outside the if block.

```
a = 0
if a < 1:
  print("a < 1")
print("great!")
```

Indention

- Statements inside an **`if`** block must all have **one level of indention**.
- There is **no indention-size restriction**; all we need is to make it **consistent** for all statements inside the same block.
- Which are good and which are bad?

```
a = 0
if a < 1:
 print("a < 1")
 print("great!")
```

```
a = 0
if a < 1:
    print("a < 1")

print("great!")
```

```
a = 0
if a < 1:
    print("a < 1")
  print("great!")
```

```
a = 0
if a < 1:
 print("a < 1")

print("great!")
```

The `if-else` statement

```
if condition:
  statements 1
else:
  statements 2
```

- In many cases, we hope that conditional on whether the condition is true or false, we do different sets of statements.
- This is done with the `if-else` statement.
  - Do ***statements 1*** if ***condition*** returns **`true`**.
  - Do ***statements 2*** if ***condition*** returns **`false`**.
- An **`else`** must have an associated **`if`**.

The `if-else` statement

- The previous example may be improved with the **`else`** statement:

```
income = float(0)
tax = float(0)

print("Please enter your income:")
income = float(input())

if income <= 10000:
  tax = 0.02 * income
if income > 10000:
  tax = 0.08 * (income - 10000) + 200

print("Tax amount: $" + str(tax))
```

```
income = float(0)
tax = float(0)

print("Please enter your income:")
income = float(input())

if income <= 10000:
  tax = 0.02 * income
else:
  tax = 0.08 * (income - 10000) + 200

print("Tax amount: $" + str(tax))
```

The `if-else` statement

- Is this right or wrong?

```
income = float(0)
tax = float(0)

print("Please enter your income:")
income = float(input())

if income <= 10000:
  tax = 0.02 * income
  else:
    tax = 0.08 * (income - 10000) + 200

print("Tax amount: $" + str(tax))
```

The Boolean data type

- We have introduced three data types: integer, float, and string.
  - Variables of these types can be created by **`int()`**, **`float()`**, and **`str()`**.
- Another common data type is the **Boolean data type** (布林資料型態).
  - There are only two possible values: **true** and **false**.
  - A Boolean variable is also called a binary variable.
- Boolean variables can be created by **`bool()`**.
  - One may also assign **`True`** or **`False`** to a variable.

```
a = False
print(a)
print(type(a))
```

```
a = True
print(a)
print(type(a))
```

```
a = bool()
print(a)
print(type(a))
```

The Boolean data type

- Note that **`bool()`** gives us **`False`**.
- In Python (and many other modern languages):
  - False means **0**.
  - True means **not 0**.
- This explains the following program:

```
a = bool(0)
print(a)

a = bool(123)
print(a)

a = bool(-4.8)
print(a)
```

Comparison operators

- A **comparison operator** (比較運算子) compares two operands and returns a **Boolean** value.
  - `>`: bigger than
  - `<`: smaller than
  - `>=`: not smaller than
  - `<=`: not bigger than
  - `==`: equals
  - `!=`: not equals

```
a = 10
b = 4
s = "123"

print(a < b)
print(len(s) != b)
print((a + 2) == (b * 3))
```

Comparison vs. assignment

- Note that to compare whether two values are identical, we use `==`, not `=`.
  - `==` is a comparison operator.
  - `=` is an assignment operator.
- `=` assigns the **value at its right** to the **variable at its left**.
  - If a variable is at its right, its value is used.
  - If it is not a variable at its left, there is a syntax error.

```
a = 10
b = 4
c = a

print(c == a)
```

```
4 = d
a + b = 4
```

Comparison vs. assignment

- Do not get confused by `=` and `==`:

```
a = 10
a = a + 2

print(a == 12)
```

- In summary:
  - Read `==` as "**equals**": `if a == b + 2` is asking whether **a equals `b + 2`**.
  - Read `=` as "**becomes**": `a = a + 2` means **`a` becomes `a + 2`**.

Programming for Business Computing

| Conditionals (2)

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Nested `if-else` statement

- An **`if`** or an **`if-else`** statement can be **nested** (巢狀的) in an **`if`** block.
  - In this example, if both conditions are true, ***statements A*** will be executed.
  - If ***condition 1*** is true but ***condition 2*** is false, ***statements B*** will be executed.
  - If ***condition 1*** is false, ***statements C*** will be executed.
- An **`if`** or an **`if-else`** statement can be nested in an **`else`** block.
- We may do this for any level of **`if`** or **`if-else`**.

```
if condition 1:
  if condition 2:
    statements A
  else:
    statements B
else:
  statements C
```

Example of nested `if-else` statements

- Given three integers, how to find the smallest one?
- Nested **`if-else`** helps:
- Some questions:
  - What will happen if there are multiple smallest values?
  - Are there better implementations?

```
a = int(input())
b = int(input())
c = int(input())

if a <= b:
  if a <= c:
    print(a, "is the smallest")
  else:
    print(c, "is the smallest")
else:
  if b <= c:
    print(b, "is the smallest")
  else:
    print(c, "is the smallest")
```

Two different implementations

```
min = 0
if a <= b:
  if a <= c:
    min = a
  else:
    min = c
else:
  if b <= c:
    min = b
  else:
    min = c
print(min, "is the smallest")
```

```
min = c
if a <= b:
  if a <= c:
    min = a
else:
  if b <= c:
    min = b
print(min, "is the smallest")
```

Indention matters

- In Python, an **`else`** will only be paired to the **`if`** **at the same level**.
- What does the following two problems mean?

```
if a == 10:
  if b == 10:
    print("a and b are both ten.\n")
else:
  print("a is not ten.\n")
```

```
if a == 10:
  if b == 10:
    print("a and b are both ten.\n")
  else:
    print("a is not ten.\n")
```

The ternary if operator

- In many cases, what to do after an **`if-else`** selection is simple.
- The **ternary if operator** (三元條件運算子) can be helpful in this case.

```
operation A if condition else operation B
```

  - If ***condition*** is true, do ***operation A***; otherwise, ***operation B***.

- Let’s modify the previous example:

```
if a <= b:
  min = a if a <= c else c
else:
  min = b if b <= c else c
```

The ternary if operator

- **Parentheses are helpful** (though not needed):

```
if a <= b:
  min = a if (a <= c) else c
else:
  min = b if (b <= c) else c
```

```
if a <= b:
  min = (a if (a <= c) else c)
else:
  min = (b if (b <= c) else c)
```

- Ternary if operators can also be nested (but **not suggested**):

```
min = (a if a <= c else c) if a <= b else (b if b <= c else c)
```

```
min = (a if a <= c else c) if (a <= b) else ((b if b <= c else c))
```

The `else-if` statement

- An **`if-else`** statement allows us to respond to one condition.
- When we want to respond to more than one condition, we may put an **`if-else`** statement in an **`else`** block:

```
if a < 10:
  print("a < 10.")
else:
  if a > 10:
    print("a > 10.")
  else:
    print("a == 10.")
```

- For this situation, people typically combine the second **`if`** behind **`else`** to create an **`else-if`** statement:

```
if a < 10:
  print("a < 10.")
elif a > 10:
  print("a > 10.")
else:
  print("a == 10.")
```

The `else-if` statement

- An **`else-if`** statement is generated by using two nested **`if-else`** statements.
- It is logically fine if we do not use **`else-if`**.
- However, if we want to respond to many conditions, using **`else-if`** greatly enhances the **readability** of our program.

```
if month == 1:
  print("31 days")
elif month == 2:
  print("28 days")
elif month == 3:
  print("31 days")
elif month == 4:
  print("30 days")
elif month == 5:
  print("31 days")
# ...
```

```
if month == 1:
  print("31 days")
else:
  if month == 2:
    print("28 days")
  else:
    if month == 3:
      print("31 days")
    else:
      # ...
```

Programming for Business Computing

| Logical Operators

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Logical operators

- In some cases, the condition for an **`if`** statement is complicated.
  - If I am hungry **and** I have money, I will buy myself a meal.
  - If I am not hungry **or** I have no money, I will not buy myself a meal.
- We may use **logical operators** (邏輯運算子) to combine multiple conditions.
- We have three logical operators: **`and`**, **`or`**, and **`not`**.
- There is a **precedence rule** (優先權規則) for operators.
  - You may find the rule in the textbook.
  - You do not need to memorize them: Just use **parentheses**.

Logical operators: and

- The "and" operator operates on **two conditions**.
- It returns true if **both** conditions are true. Otherwise it returns false.
  - `(3 > 2) and (2 > 3)` returns `False`.
  - `(3 > 2) and (2 > 1)` returns `True`.
- When we use it in an `if` statement, the grammar is:

```
if condition 1 and condition 2:
  statements
```

Logical operators: and

- As an example:

```
a = int(input())
b = int(input())
c = int(input())

if a < b and b < c:
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

Logical operators: and

- An "and" operation can replace a nested **`if`** statement.

```
a = int(input())
b = int(input())
c = int(input())

if a < b and b < c:
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

```
a = int(input())
b = int(input())
c = int(input())

if a < b:
  if b < c:
    print("b is in between a and c")
  else:
    print("b is outside a and c")
else:
  print("b is outside a and c")
```

Logical operators: and

- Sometimes conditions may be combined without a logical operator:

```
if a < b < c:
  print("b is in between a and c")
```

- Nevertheless, avoid weird expressions (unless you know what you are doing):

```
if a < b < c > 10: # not good
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

- Each condition must be complete by itself:

```
if b > a and < c: # error!
  print("a is between 10 and 20")
```

Logical operators: or

- The "or" operator returns true if **at least** one of the two conditions is true. Otherwise it returns false.
  - `(3 > 2) or (2 > 3)` returns `True`.
  - `(3 < 2) or (2 < 1)` returns `False`.
- When the or operator is used in an `if` statement, the grammar is

```
if condition 1 or condition 2:
  statements
```

Logical operators: or

- How about

```
if condition 1 or condition 2 or condition 3:
   statements
```

- How about

```
if condition 1 or condition 2 and condition 3:
   statements
```

Logical operator: not

- The "not" operator returns the **opposite** of the condition.
  - `not (2 > 3)` returns `True`.
  - `not (2 > 1)` returns `False`.
- It may be used when naturally there is nothing to do in the `if` block:

```
key = input("continue? ")

if key == "y" or key == "Y":
  print() # to avoid error
else:
  print("Game over!")
```

```
key = input("continue? ")

if not (key == "y" or key == "Y"):
  print("Game over!")
```

Programming for Business Computing

| Formatting a Program

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Formatting a program

- Maintaining the program in a good **format** is very helpful.
- While each programmer may have her own programming style, there are some general guidelines for Python.
  - Add proper white spaces and empty lines.
  - Give variables understandable names.
  - Write comments.

Write spaces and empty lines

- Some suggestions about white spaces and empty lines are useful.
  - Add **two white spaces** around a binary operator.
  - Add a white space after each comma.
  - Use **empty lines** to separate groups of codes.
- Which one do you prefer?

```
print("Please enter one number:")
num1 = int(input())
print("Please enter another number:")
num2 = int(input())

print("The sum is", num1 + num2)
```

```
print("Please enter one number:")
num1 =int(input())
print("Please enter another number:")
num2= int(input())
print("The sum is",num1 + num2)
```

Variable declaration

- When declare variables:
  - Give variables **understandable names**.
- Which one do you prefer?

```
dice1 = int(input())
dice2 = int(input())

sum = dice1 + dice2

print(sum)
```

```
a = int(input())
b = int(input())

c = a + b

print(c)
```

Comments

- **Comments** (註解) are programmers' **notes** and will be ignored by the compiler.
- In Python, there are two ways of writing comments:
  - A single line comment: Everything following a **#** in the same line are treated as comments.
  - A block comment: Everything within a pair of **"""** (may across multiple lines) are treated as comments.

```
"""
Ling-Chieh Kung's work
for the first lecture
"""

print("Hello World! \n") # the program terminates correctly
```

- Hotkeys are very helpful. Use them!