Programming for Business Computing

| Computers, Types, and Precision

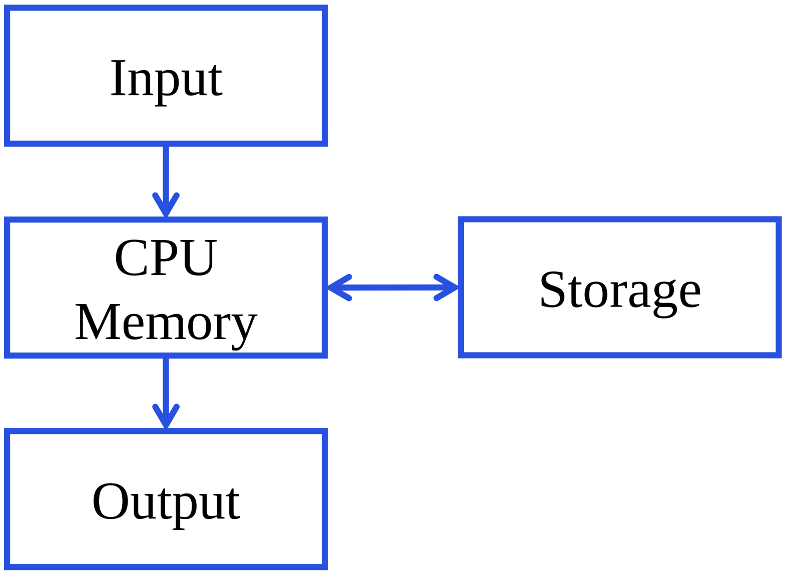
Ling-Chieh Kung

Department of Information Management

National Taiwan University

Computers

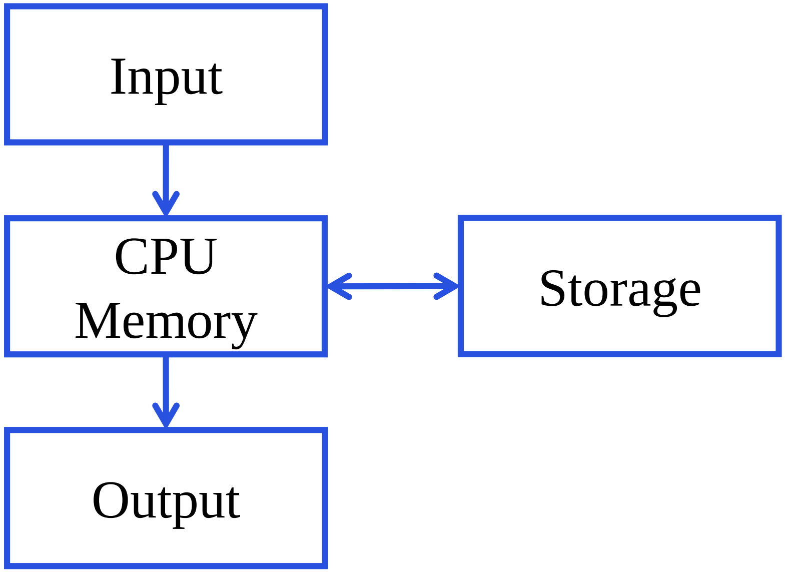
- In a modern computer:
- "**Input**" includes keyboards, mice, touch screens, microphones, etc.
- "**Output**" include screens, speakers, printers, etc.
- "**Storage**" means non-volatile storage, such as hard discs, CDs, DVDs, flash drives, etc.



- "**CPU & Memory**":
  - "CPU" (central processing unit, 中央處理器) is where arithmetic operations are done.
  - "Memory" (記憶體) is a volatile storage space.

Programs (程式)

- A **program** is a file containing source codes.
  - It is stored in "storage".
- When we execute/run a program:
  - We create **variables** in "memory" to store **values**.
  - We move values into "CPU" for **arithmetic operations**, and then move the results back to "memory".



- We may do more:
  - We (probably) **read** from "input" and **write** to "output".
  - We (probably) **read** from "storage" and **write** to "storage".

Variables and values

- When we declare a **variable** (變數), the **operating system** (OS; 作業系統) allocates a space in memory for that variable.
  - Later **values** (值) can be stored there.
  - That value can be read, written, and overwritten.
- The OS records four things for each variable:
  - Memory address (記憶體位址).
  - Name (also called “identifier”).
  - Value.
  - **Type** (型態).

When we execute this program

```
num1 = 13
num2 = 4
print(num1 + num2)
```

	Address	Identifier	Value
(3)	0x20c630	(no name)	17
(1)	0x20c648	num1	13
(2)	0x22fd4c	num2	4

Memory

(4)

```
17
```

Console

Types

- A variable's type is **automatically** determined by Python according to the type of the initial value.
  - In some other programming languages, the programmer must determine it.
  - E.g.,

```
num1 = 13
num2 = 4.13
str1 = "52"
```

    makes **num1** an **integer** (整數), **num2** a **floating-point number** (浮點數), and **str1** a **string** (字串).
- These are the most important three types at this moment:
  - An integer is an integer.
  - A string is a sequence of characters.
  - What is a floating-point number?

Integers

- A computer stores values in a **binary system**.
- A binary number $a_3a_2a_1a_0$, where $a_i \in \{0, 1\}$ for all i,
- equals the decimal number $8a_3 + 4a_2 + 2a_1 + a_0$.

$$\boxed{a_3}\boxed{a_2}\boxed{a_1}\boxed{a_0} \Rightarrow 8a_3 + 4a_2 + 2a_1 + a_0$$

  - See the table at the right for a typical mapping.
  - With four **bits**, a binary variable may store 16 values.
- Today common lengths of an integer are 16 bits, 32 bits, 64 bits, 96 bits, 128 bits, etc.
  - 1 byte = 8 bits.
- In general, with n bits, a binary number $a_{n-1}a_{n-2}\cdots a_1a_0$ equals the decimal number $\sum_{i=0}^{n-1} 2^i a_i$.

Decimal value	Binary value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
$\cdots$	$\cdots$
15	1111

Signed integers

- Integers may be positive, zero, or negative.
- To represent negative numbers, we use **the first bit** to denote the **sign**.
- A binary number $a_3a_2a_1a_0$ equals the decimal number $(-1)^{a_3} \times (4a_2 + 2a_1 + a_0)$ in one mapping system.

$$\boxed{a_3}\boxed{a_2}\boxed{a_1}\boxed{a_0} \Rightarrow (-1)^{a_3} \times (4a_2 + 2a_1 + a_0)$$

Decimal value	Binary value
0	0000
1	0001
2	0010
3	0011
…	…
–5	1101
–6	1110
–7	1111

Integers in Python

- Integers in Python are by default signed.
  - They can represent negative values.
- To create an integer with an **initial value**, simply do it:

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

  - The function **`type()`** returns the type of a given variable.
- To create an integer without an initial value, use the function **`int()`**.

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

Floating-point numbers

- To represent **fractional numbers**, most computers use **floating-point numbers**.
- The rough idea is:

a_5	a_4	a_3	a_2	a_1	a_0

$$(-1)^{a_5} \times (2a_1 + a_0) \times 2^{(-1)^{a_4} \times (2a_3 + a_2)}$$

- For example,

0	1	1	0	1	1

$$3 \times 2^{-2} = 0.75$$

- Moreover, the "binary point" may "float" to make the mapping flexible.
  - To represent more values or increase precision.
  - This is why a fractional number is called a floating-point number.
- The true standard for floating-point numbers is more complicated.

Floating-point numbers in Python

- A floating-point number (or simply “a float”) in Python are by default signed.
- To create a float with an initial value, simply do it:

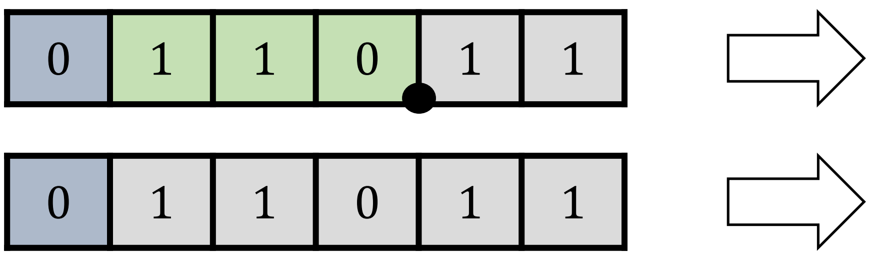
```
i = 52.0
print(i)
print(type(i))
```

- To create a float without an initial value, use the function **`float()`**.

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

Memory allocation

- When we declare a variable, its type matters.
  - The OS understands its value based on its type.
  - An integer and a floating-point number represent **different values** even if they store the same sequence of bits.



$-3 \times 2^{-2} = 0.75$

27

- This is why each variable needs to have a **type**.

Characters (字元)

- A computer cannot store **characters** directly.
- It represents characters by encoding each character into an integer.
- In most PCs, we use the **ASCII code**.
  - ASCII = American Standard Code for Information Interchange.
  - It uses **one byte** (–128 to 127) to represent English letters, numbers, symbols, and special characters (e.g., the newline character).
  - E.g., "0" is 48, "A" is 65, "a" is 97, etc.
  - It does not represent, e.g., Chinese characters.

Characters

- Try this:

```
c = 52
c_as_chr = chr(c)
print(c_as_chr)
```

  - An integer **`c`** is created and assigned 52. .
  - The corresponding character “4” in the ASCII table is printed out.
  - **`c`** is an integer (**`int`**), but **`c_as_chr`** is a character (**`chr`**).

Characters/strings in Python

- To create a character with an initial value, simply do it:

```
c = "52"
print(c)
print(type(c))
```

  - Note that the type is “str”, which means a string.
- A **string** is a sequence of characters.
- In fact, even a single character is considered a string (of length 1) in Python.

```
c = "1"
print(c)
print(type(c))
```

String operations in Python

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

```
s = "52"
print(s)
print(len(s))
```

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

```
s1 = "52"
s2 = " is good"
s = s1 + s2
print(s)
print(len(s))
print(s2 + s1)
print(len(s2 + s1))
```

Non-English characters and symbols

- To represent Chinese (and other non-English) characters, we need other encoding standards.
  - Common standards include UTF-8, Big-5, etc.
- Special symbols (like 「, 、, ～, etc.) also need to be encoded.
  - English characters and symbols are all **halfwidth** (半形).
  - All **fullwidth** (全形) symbols are non-English symbols.
- We will deal with Chinese in the future.

Programming for Business Computing

| Iterations (1)

Ling-Chieh Kung

Department of Information Management

National Taiwan University

The `while` statement

- In many cases, we want to repeatedly execute a set of codes.
- One way to implement **repetition** is to write **a `while` loop** (while 迴圈).
- Guess what do these programs do?

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i = i + 1

print(sum)
```

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

- **`while`** is nothing but an **`if`** that **repeats**.
  - The statements in a while block are repeated if the condition is satisfied.

Modifying loop counters

- We may need to add 1 to or subtract 1 from a **loop counter** (迴圈計數器).
- Binary **self assignment** operators (e.g., **+=**) may help.

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i = i + 1

print(sum)
```

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i += 1

print(sum)
```

```
sum = 0
i = 1

while i <= 100:
  sum += i
  i +=  1

print(sum)
```

Example

- Given an integer n, is $n = 2^k$ for some integer $k \geq 0$?

```
n = int(input())
k = 0
m = 1

while n > m:
  m *= 2
  k += 1
  # print(m, k)

if m == n:
  print(n, "is 2 to the power of", k)
```

Infinite loops

- An **infinite loop** (無窮迴圈) is a loop that does not terminate.

```
n = int(input())
k = 0
m = 1

while n != m:
  m *= 2
  k += 1

if m == n:
  print(n, "is 2 to the power of", k)
```

- In many cases an infinite loop is a **logical error** made by the programmer.
  - When it happens, check your program.

`break` and `continue`

- When we implement a repetition process, sometimes we need to further change the flow of execution of the loop.
- A **`break`** statement brings us to **exit the loop** immediately.
- When **`continue`** is executed, statements after it in the loop are **skipped**.
  - The looping condition will be checked immediately.
  - If it is satisfied, the loop starts from the beginning again.

Example

- Which of the following programs work?

```
n = int(input())
m = n
k = 0

while m > 1:
  if m % 2 != 0:
    break
  m //= 2
  k += 1

if m == 1:
  print(n, "is 2 to the power of", k)
```

```
n = int(input())
m = n
k = 0

while m > 1:
  if m % 2 != 0:
    continue
  m //= 2
  k += 1

if m == 1:
  print(n, "is 2 to the power of", k)
```

`break` and `continue`

- The effect of **`break`** and **`continue`** is just on **the current level**.
  - If a **`break`** is used in an inner loop, the execution jumps to the outer loop.
  - If a **`continue`** is used in an inner loop, the execution jumps to the condition check of the inner loop.
- What will be printed out at the end of this program?

```
a = 1
b = 1
while a <= 10:
  while b <= 10:
    if b == 5:
      break
    print(a * b)
    b += 1
  a += 1
print(a)  # ?
```

Infinite loops with a `break`

- We may intentionally create an infinite loop and terminate it with a **`break`**.
  - E.g., we may wait for an “exit” input and then leave the loop with a **`break`**.

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

```
while True:
  # do something
  exit = input("Press y or Y to exit: ")
  if exit == "y" or exit == "Y":
    break
```

Infinite loops with a `break`

- The above mentioned technique is widely used to eliminate redundant codes.

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

- Redundancy introduces potential **inconsistency**.
- In some other languages, this technique is offered as a “do-while loop”.
  - In Python, just do it by yourself.

`break` and `continue`

- Using **`break`** gives a loop **multiple exits**.
  - It becomes harder to track the flow of a program.
  - It becomes harder to know the state after a loop.
- Using **`continue`** highlights the need of **getting to the next iteration**.
  - Having too many continue still gets people confused.
- Be careful **not to hurt the readability** of a program too much.

Programming for Business Computing

Iterations (2)

Ling-Chieh Kung

Department of Information Management

National Taiwan University

The `for` statement

- Another way of implementing a loop is to use **a `for` loop** (for 迴圈).

```
for variable in list:
    statements
```

- The typical way of using a for statement is:
  - ***variable***: A variable called the loop counter.
  - ***list***: A list of variables that will be "traversed."
  - ***statements***: The things that we really want to do.
- In each iteration, ***variable*** will take a value in ***list*** (from the first to the last).

Example

- To create a list, simply list them:

```
for i in 1, 2, 3:
  if i % 2 != 0:
    print(i)
```

```
a = 1
b = 2
c = 3

for i in a, c, b:
  print(i)
```

- A string can also be treated as a list.
  - Each character will be considered in each iteration.

```
str = "abwyz"

for i in str:
  print(i + "1")
```

`range()`

- The `range()` function is useful in creating a list of integers.
  - If n is input into **`range()`**, a list of integers $0, 1, 2, \ldots, n - 1$ is returned.
  - If m and n are input into **`range()`**, a list of integers $m, m + 1, m + 2, \ldots, n - 1$ is returned.
  - If m, n, and k are input into **`range()`**, a list of integers $m, m + k, m + 2k, \ldots$ is returned, where the last integer plus k is greater than $n - 1$.
- More details about list will be introduced later in this semester.
  - For now, let's just use it in a **`for`** loop.

`for` vs. `while`

- Let's calculate the sum of 1 + 2 + … + 100:
  - We used **`while`**. How about **`for`**?

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i = i + 1

print(sum)
```

- To use **`for`**:
  - We first prepare a list of values 1, 2, …, and 100.
  - Then we sum them up.

```
sum = 0

for i in range(1, 101):
  sum = sum + i

print(sum)
```

Modifying the loop counter?

- What will be the outcome of this program?

```
sum = 0

for i in range(1, 11):
  sum = sum + i
  i = i + 10

print(sum)
```

- Manual modifications of the loop counter is of no effect!

Nested loops

- Like the selection process, **loops** can also be **nested**.
  - Outer loop, inner loop, most inner loop, etc.
- Nested loops are not always necessary, but they can be helpful.
  - Particularly when we need to handle a **multi-dimensional** case.

Nested loops: Example 1

- Please write a program to output some integer points on an (x, y)-plane like this:

(1, 1) (1, 2) (1, 3)

(2, 1) (2, 2) (2, 3)

(3, 1) (3, 2) (3, 3)

```
for x in range(3):
  x += 1
  for y in range(3):
    y += 1
    print("(" + str(x) + ", " + str(y) + ")", end = " ")
  print()
```

- Note the **`end = " "`** in the inner **`print`**.
  - It says "do not change to a new line" but "append a white space."
  - We change to a new line only in the outer loop by printing out a newline character.
- This can still be done with only one level of loop. but using a nested loop is much easier.

Nested loops: Example 2

- Please write a program to output a multiplication table:

```
for x in range(1, 5):
  for y in range(1, 5):
    print(str(x) + " * " + str(y) + " = " + str(x * y) + ";", end = " ")
  print()
```

- How would you make the lower and upper bounds flexible?
- How would you align the outputs in the same column?

Case study: single-product pricing

- We sell a product to a small town.
- The demand of this product is $q = a - bp$:
  - a is the base demand.
  - b measures the price sensitivity of the product.
  - p is the unit price to be determined.
- Let c be the unit production cost.
- Given a, b, and c, how to solve

$$\max_{p} \, (a - bp)(p - c)$$

to find an optimal (profit-maximizing) price p^*?

Case study: single-product pricing

- Where there is an analytical solution $p^* = \frac{a+bc}{2b}$ (please consult the professors of your Economics/Calculus/Marketing courses), let's write a program to solve it.
- Let's assume that the price can only be an integer:

```
a = int(input("base demand = "))
b = int(input("price sensitivity = "))
c = int(input("unit cost = "))

max_profit = 0
optimal_price = 0
for p in range(c + 1, a // b):
  profit = (a - b * p) * (p - c)
  # print(p, profit)

  if profit > max_profit:
    max_profit = profit
    optimal_price = p

print("optimal price = " + str(optimal_price))
print("maximized profit = " + str(max_profit))
```

Case study: single-product pricing

- Note that the profit as a function of price is first increasing and then decreasing (why?).
  - Once a price results in a profit that is lower than the maximum profit, all further prices cannot be optimal.
  - We may revise our program accordingly.

```
a = int(input("base demand = "))
b = int(input("price sensitivity = "))
c = int(input("unit cost = "))

max_profit = 0
optimal_price = 0
for p in range(c + 1, a // b):
  profit = (a - b * p) * (p - c)
  # print(p, profit)

  if profit > max_profit:
    max_profit = profit
    optimal_price = p
  else:
    break

print("optimal price = " + str(optimal_price))
print("maximized profit = " + str(max_profit))
```

Good programming style

- Use the loop that makes your program the most **readable**.
- When you need to execute a loop for **a fixed number of iterations**, use a **`for`** statement with a counter declared only for the loop.
  - This also applies if you know the maximum number of iterations.
  - If the number of (maximum) number of iterations is uncertain, use **`while`**.

Programming for Business Computing

| Precision Issue of Floating-point Values

Ling-Chieh Kung
Department of Information Management
National Taiwan University

Precision can be a big issue

- Please execute the following program and try to explain the outcome:

```
import math

bad = 0
for i in range(100):
  f = pow(i, 1/2)

  if f * f != i:
    print("!!!")
    bad += 1
  else:
    print()

print("bad precision:", bad)
```

Precision can be a big issue

- Let’s understand it:

```
import math

bad = 0
for i in range(100):
  f = pow(i, 1/2)
  print(i, f * f, end = " ")

  if f * f != i:
    print("!!!")
    bad += 1
  else:
    print()

print("bad precision:", bad)
```

Precision can be a big issue

- Precision can be a big issue when we use floating-point values.
- As modern computers store values in bits, most **decimal fractional numbers** can only be **approximated**.
  - 3
  - 3.375
  - 3.4375
  - 3.4?

3	1	1	.	0	0	0	0
3.375	1	1	.	0	1	1	0
3.4375	1	1	.	0	1	1	1

- Therefore, that **`f = pow(i, 1/2)`** does not make **`f`** storing the **exact value** of square root of **`i`**. There must be some error.

Precision can be a big issue

- Remedy: "imprecise" comparisons.

```
if abs(f * f - i) > 0.0001:
    print("!!!")
    bad += 1
else:
    print()
```

- The error tolerance can be neither too large nor too small.
  - It should be set according to the property of your own problem.
- To learn more about this issue, study *Numerical Methods*, *Numerical Analysis*, *Scientific Computing*, etc.