Programming for Business Computing

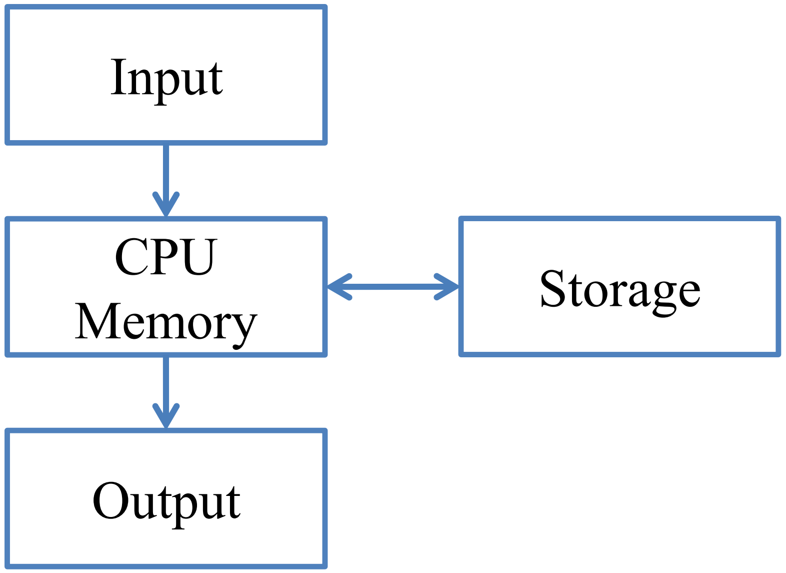
Computers, Types, and Precision

Ling-Chieh Kung

Department of Information Management
National Taiwan University

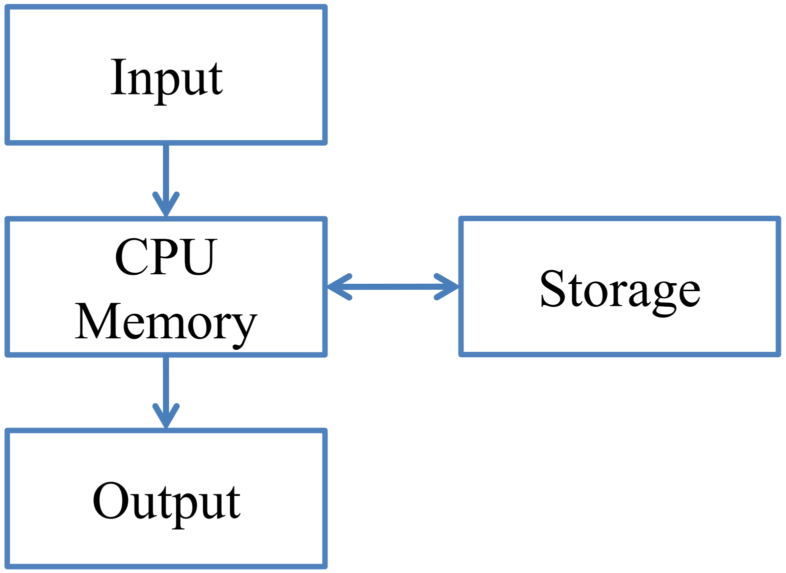
Computers

- 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) is where arithmetic operations are done.
  - "Memory" is a volatile storage space.



Programs

- 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".

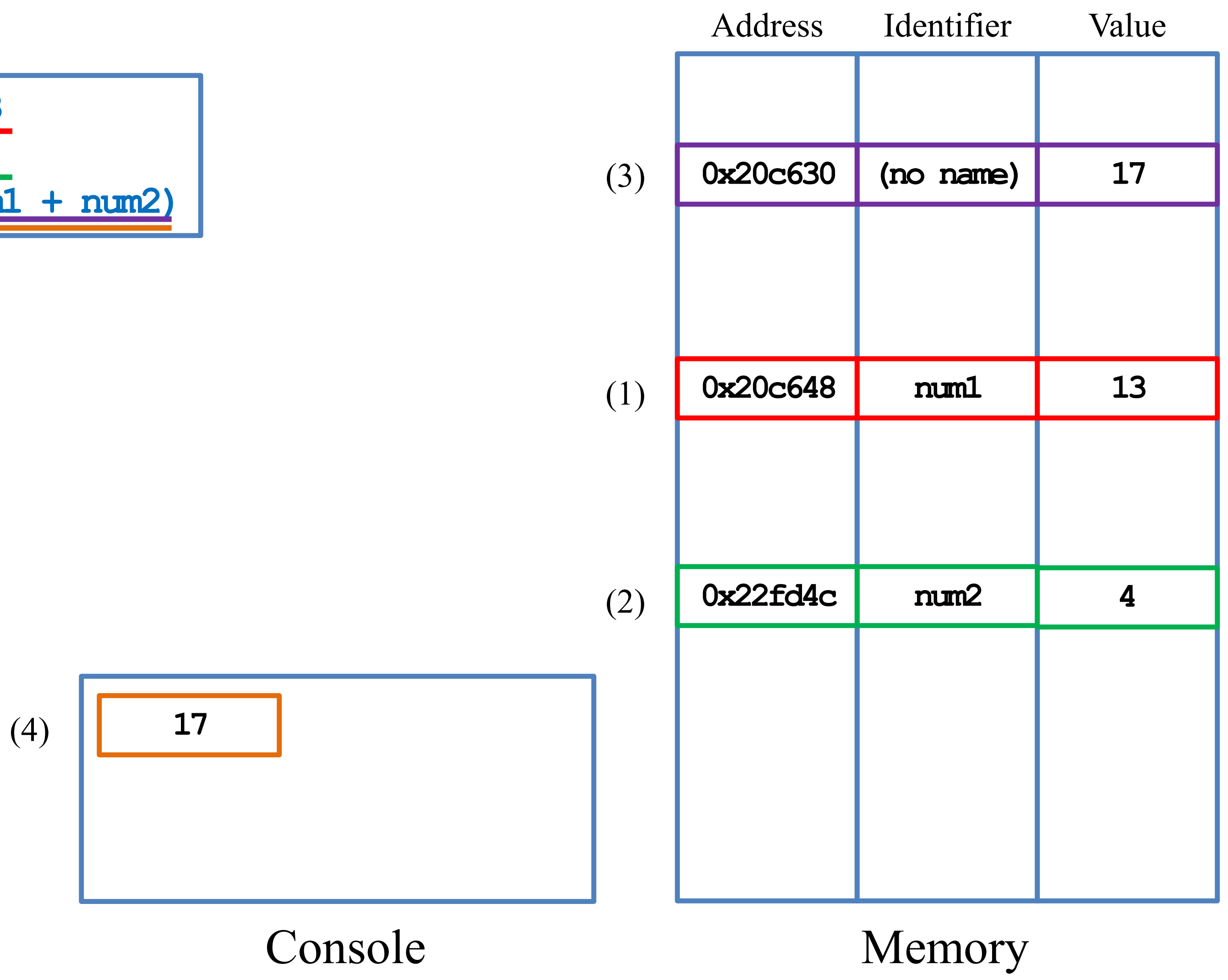


Variables and values

- When we declare a **variable**, the operating system (OS) allocates a space in memory for that variable.
  - Later **values** can be stored there.
  - That value can be read, written, and overwritten.
- The OS records four things for each variable:
  - Address.
  - Name (also called "identifier").
  - Value.
  - Type.

When we execute this program

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



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**, **num2** a **floating-point number**, and **str1** a **string**.

- 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 $8a_3 + 4a_2 + 2a_1 + a_0$.

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

  - 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
…	…
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:

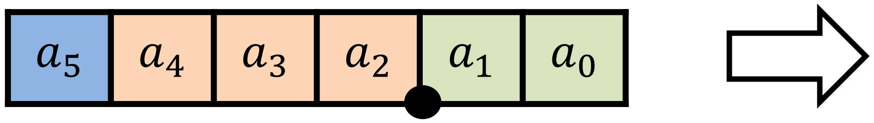
```
i = 52
print(i)
print(type(i))
```

  - 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))
```

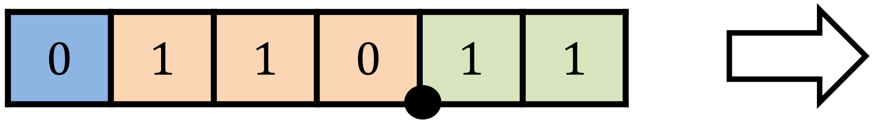
Floating-point numbers

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



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

- For example,



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

- 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:

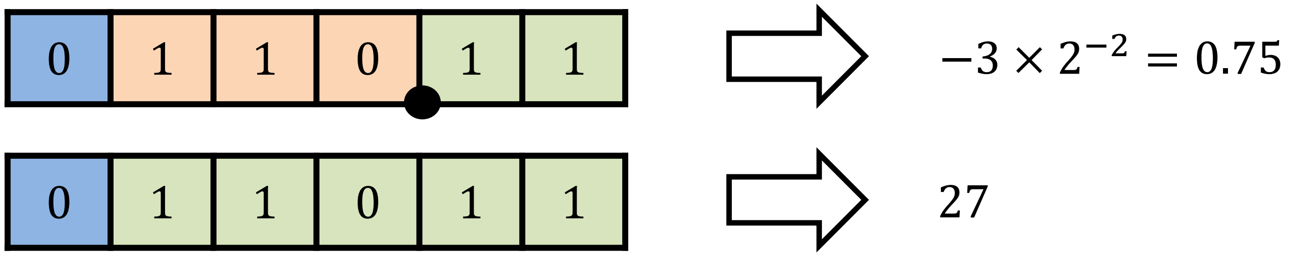
```
i = 52.0
print(i)
print(type(i))
```

- 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.



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

Characters

- 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
cAsChr = chr(c)
print(cAsChr)
```

  – 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 **`cAsChr`** 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**.
  - All **fullwidth** symbols are non-English symbols.
- We will deal with Chinese in the next course.

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**.
- 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** (**=**), 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** 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`**.
- 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**.
- 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**.

```
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**.
  - 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** 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

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** 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!