Variables and Data Types

Python Programming

Byeongjoon Noh

powernoh@sch.ac.kr

SCH SOON CHUN HYANG UNIVERSITY

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

Contents

Textbook: Chapter 1, Chapter 3, Chapter 4, Chapter 5.1~5.6

1. Introduction

Hello world!

- Printed “Hello, World!” in hello_world.py python program in the previous chapter
  - Merely prints out a string with the welcome message:

```
print('Hello World!')
```

- What is `print()`? and where does the `print()` function come from?
  - a predefined function that can be used to *print things out*, for example to the user
    - “predefined”: built into the Python environment and is understood by Python interpreter
    - ➔ the interpreter knows where to find the definition of the `print()` function which tells it what to do when it encounters the `print()` function
  - this handles a stream (sequence) of data such as letters and numbers
  - this output stream of data can be sent to an output window such as the terminal on Mac or Command Window on Windows PC

Hello world!

- The `print()` function actually tries to print whatever you give it,
  - when it is given a **string** it will print a string
  - if it is given an **integer** such as `42` it will print `42` and
  - if it is a given a **floating point** number such as `23.56` then it will print `23.56`

Interactive hello world

- Let us make our program, hello_world.py, a little more interesting
  - to ask us our name and say hello to us personally

```
print('Hello world!')
user_name = input('Enter your name: ')
print('Hello ', user_name)
```

```
Hello world!
Enter your name: John
Hello John
```

  - `user_name = input('Enter your name: ')`
    - this statement first execute another function called `input()`
      - this function is passed a string (a.k.a an argument) to use when it prompts the user for input
      - also a built-in function in Python environment
    - result is stored in the ***variable*** `user_name`

Assignment operator

- ‘=’
  - *Assignment* operator
  - between the `user_name` variable and the `input()` function;

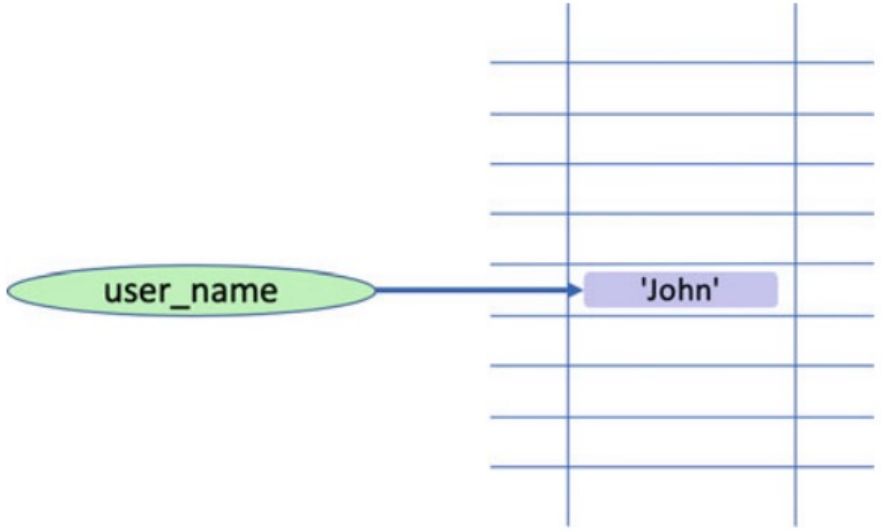
```
user_name = input('Enter your name: ')
```

  - Used to assign the value returned by the function `input()` to the variable `user_name`

2. Variables

Definition

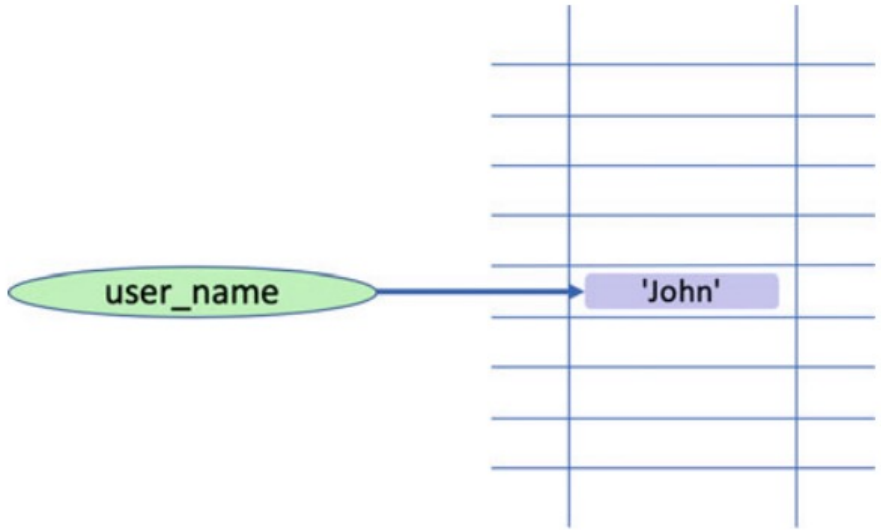
- Variable (변수): A named area of the computers' memory that can be used to hold things
  - often referred to as data
  - e.g., strings, numbers, Boolean (True/False), etc.
  - `user_name` is acting as a label for an area of memory which will hold the string entered by user
    - can refer an area of memory containing actual data



Two dimensional grid "memory" location; each location has an address associated with it

- address is unique within the memory and can be used to return to the data held at that location

Note: Memory, address, and variable



- this address is often referred to as **memory address** of the data
  - ➔ this memory address that is actually held in the variable `user_name`
  - ➔ `user_name` is shown as pointing to the area in memory containing the string 'John'

```
print('Hello ', user_name)
```

- if we want to get hold of the name entered by the user in another statement, we can do by referencing the variable `user_name`

Variable

- Let's modify hello_world.py
  - to ask the user for the name of their best friend and print out a welcome message to that best friend

```
print('Hello, world')
name = input('Enter your name: ')
print('Hello', name)
name = input('What is the name of your best friend: ')
print('Hello Best Friend', name)
```

```
Hello world!
Enter your name: John
Hello John
What is the name of your best friend: Denise
Hello Best Friend Denise
```

  - ➔ Because the area of memory that previously held the string 'John' now holds the string 'Denise'

Variable declaration

- Variable in Python is not restricted to holding a string; ‘John’ and ‘Denise’
  - can also hold other types of data such as numbers or the values (True/False)

```
my_variable = 'John'
print(my_variable)
my_variable = 42
print(my_variable)
my_variable = True
print(my_variable)
```

```
John
42
True
```

- Note: Python has no command for declaring a variable

Variable declaration

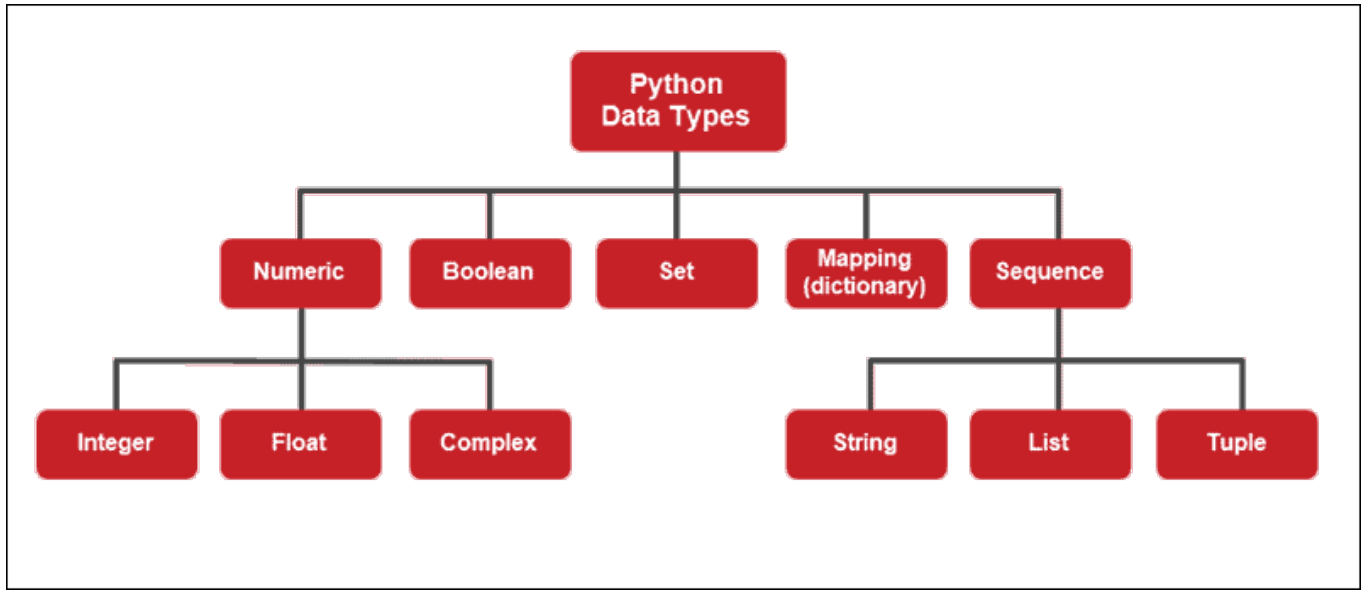
- Provide the various variable declaration methods in Python

```
x, y, z = "Orange", "Banana", "Cherry"
print(x, y, z)

x, y, z = 5, 3.7, "Cherry"
print(x, y, z)

x = y = z = "Orange"
print(x, y, z)
```

```
Orange Banana Cherry
Orange Orange Orange
5 3.7 Cherry
```

Data types

- Primary data types
  - Numeric; Integer, Float, Complex
  - Boolean; True or False
  - String



- Collection types
  - Four classes in Python that provide container; that is data types of holding collections of other objects
  - List
  - Tuple
  - Set
  - Dictionary

Get the type

- Use `type()` function to obtain data type

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

```
<class 'int’>
<class 'str'>
```

Naming rules

- Rules for Python variable names **should**:
  - only contain alpha-numeric characters and underbar; (A-Z, a-z, 0-9, and _)
  - start with a letter or the underbar character
  - cannot start with a number
  - case-sensitive; age, Age and AGE are three different variables
  - cannot use “keyword”
    - if, for, return, def, …
    - How to obtain all keywords list in Python;

```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Naming rules

- Legal variable names:

```
myvar = "John"
my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

- Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Note: Naming conventions

- Variable names
  - e.g. `user_name` and `my_variable` ← Snake naming convention
  - Both these variable names are formed of a set of characters with an underbar between 'words'

- Highlight a very widely used naming convention in Python; that variable names **will**:
  - be all lowercase
  - be in general more descriptive than variable names (e.g. `a` or `b`)
    - although there are some exceptions such as the use of variables `i` and `j` in looping constructs
    - with individual words separated by underscores as necessary to improve readability

Note: Naming conventions

- Camel case
  - Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

- Pascal case
  - Each word starts with a capital letter:

```
MyVariableName = "John"
```

- Snake case
  - Each word is separated by an underbar character with all lowercase

```
my_variable_name = "John"
```

Comments in code

- To add comments to code to help anyone reading the code to understand what the code does, what its intent was, any design decisions the programmer made etc.
- Comments are ignored by the Python interpreter
  - they are not executable code
- Comment is indicated by the '#' character in Python

```
# This is a comment
name = input('Enter your name: ')
# This is another comment
print(name) # this is a comment to the end of the line
```

In class practice

- P02-01 다양한 방법으로 여러가지 변수들을 선언하고 해당 변수들에 값을 할당하는 프로그램을 작성해보세요.
  - requirements
    - 변수의 이름은 Python variable naming rule을 따를 것
    - 정수, 부동소수점, 문자열 등 다양한 변수를 다루어 볼 것
    - 한 번에 여러 변수를 선언/값 할당 해볼 것

3. Strings

What are Strings?

- String: a series, or sequence, of characters in order
  - character: anything you can type on the keyboard in one keystroke
    - a letter ‘a’, ‘b’, ‘c’, or a number ‘1’, ‘2’, ‘3’
    - a special character ‘₩’, ‘[‘, ‘$’, etc.
    - a space ‘ ’ (although it does not have a visible representation)
  - strings are immutable;
    - once a string has been created it cannot be changed

```
original_string = "Hello"
original_string[0] = "J"
```

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
      Cell In[9], line 2
      1 original_string = "Hello"
----> 2 original_string[0] = "J"
TypeError: 'str' object does not support item assignment
```

Note: Debugging

```
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
      Cell In[9], line 2
      1 original_string = "Hello"
----> 2 original_string[0] = "J"
TypeError: 'str' object does not support item assignment
```

- **Runtime errors**
  - if something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback
  - **NameError**: trying to use a variable that doesn't exist in the current environment
  - **TypeError**: there are several possible causes
    - trying to use a value improperly; indexing a string, list, or tuple with something other than an integer
    - a mismatch between the items in a format string and the items passed for conversion
    - passing the wrong number of arguments to a function

Note: Debugging

- **Runtime errors**
  - **KeyError**: trying to access an element of a dictionary using a key that the dictionary does not contain
  - **IndexError**: using to access a list, string, or tuple is greater than its length minus one
  - **AttributeError**: Triggered when an attribute reference or assignment fails, such as trying to access a method or property that does not exist for an object
  - **ZeroDivisionError**: As the name suggests, it occurs when you try to divide a number by zero
  - **IOError**: Raised when an I/O operation (like opening a file) fails for an I/O-related reason (e.g., "file not found")
  - **ImportError**: Occurs when an imported module or object cannot be found
  - **ModuleNotFoundError**: A specific case of ImportError, raised when a module cannot be found
  - etc.

Representing strings

- Single quote character ‘’
  - to define the start and end of a string
  - double quotes (“”) are also valid

```
'Hello'
'Hello World'
'Hello Andrea2000'
'To be or not to be that is the question!’
"Double quotes are also fine"
```

  - cannot mix the two styles of start and end strings; single quote and double quote

- Useful if your string need to contain one of the other type of string delimiters
  - single quote can be embedded in a string defined using double quotes and vice versa

```
print("It's the day")
print('She said "hello" to everyone')
```

Representing strings

- Triple quotes
  - support multi-line strings;

```
z = """
Hello
 World
"""
print(z)
```

- Empty string
  - has no characters in it
  - defined as single quote followed immediately by a second single quote with no gap between them

```
empty_string = ''
```

What can you do with strings?

- In python terms this means what operations or functions are their available or built-in that you can use to work with strings
  - concatenation, length, accessing, counting, replacing, splitting, etc.

String concatenation

- Concatenation
  - merge two strings together
  - using the ‘+’ operator
    - an operator is an operation or behaviour that can be applied to the types involved
  - take one string and add it to another string to create a new third string

```
string_1 = 'Good'
string_2 = " day"
string_3 = string_1 + string_2
print(string_3)
print('Hello ' + 'World')
```

```
Good day
Hello World
```

  - each string is defined with single quotes and double quotes, respectively, but does not matter here

String concatenation

- To concatenate a string and some other types using ‘+’ concatenation operator

```
msg = 'Hello Lloyd you are ' + 21
print(msg)
```

  - get an error message indicating that you can only concatenate string with string not integers with strings
    - ➔ Converting other types into strings

```
msg = 'Hello Lloyd you are ' + str(21)
print(msg)
```

```
Hello Lloyd you are 21
```

Length of a string

- Useful to know how long a string is
    - if you are putting a string into a user interface you might need to know how much of the string will be displayed within a field

- To find out the length of a string in Python, use `len()` function

```
print(len(string_3))
```

```
8
```

Accessing a character

- As a string is a fixed sequence of letters, it is possible to use square brackets [ ], and an index (or position) to retrieve a specific character from within a string
  - should note that strings are indexed from 0 (zero based indexing)

```
my_string = 'Hello World'
print(my_string[4])
```

```
o
```

  - stating [4] indicates that we want to obtain the fifth character in the string, which in this case is the letter ‘o’

Quiz

- What is the output of the following code?

```
p = 'Love for Programming'
print(p[6], p[4], p[5])
```

  - a) Error
  - b) e f
  - c) e
  - d) o f

Quiz

- What is the output of the following code?

```
msg = 'programming'
print(msg[-0])
```

  - a) Error
  - b) p
  - c) g
  - d) Blank output

Accessing a character

- Accessing a subset of string
  - to obtain a subset of the original string, often referred to as a substring
  - Use the square brackets notation but using ':' to indicate the start and end points of substring
    - Syntax: `string[start:stop:step]`
    - `start` (optional) indicates start index
    - `stop` (optional) indicates stop+1 index
    - `step` (optional) indicates step size or stride between each character in substring

```
my_string = 'Hello_World'
print(my_string[1:5]) # from index 1 to 4
print(my_string[:5]) # from start to index 4
print(my_string[2:]) # from index 2 to the end
```

```
ello
Hello
llo_World
```

Accessing a character

- Accessing a subset of string

```
my_string = 'Hello_World'
print(my_string[::]) # the entire string (slice operation)
print(my_string[0:10:2]) # from 0 to 9 step by 2
print(my_string[1:11]) # from 1 to 10 (step by 1, default)
print(my_string[10:0:-1]) # from 10 to 1 step by -1, reverse
print(my_string[::-1]) # reverse the entire string
```

```
Hello_World
HloWr
ello_World
dlroW_olle
dlroW_olleH
```

Repeating strings

- Use the '*' operator with strings
  - to repeat the given string a certain number of times
  - this generates a new string containing the original string repeated *n* number of times

```
print('*' * 10)
print('Hi' * 10)
```

```
**********
HiHiHiHiHiHiHiHiHiHi
```

Splitting strings

- To split a string up into multiple separate string based on a specific character such as a space or a comma
  - it is a very common requirement to handle data
- Use `split()` function

```
title = 'The Good, The Bad, and the Ugly'
print('Source string:', title)
print('Split using a space')
print(title.split(' '))
print('Split using a comma')
print(title.split(','))
```

```
Source string: The Good, The Bad, and the Ugly
Split using a space ['The', 'Good,', 'The', 'Bad,', 'and', 'the', 'Ugly’]
Split using a comma ['The Good', ' The Bad', ' and the Ugly']
```

  - result format is a list

Counting strings

- To find out how many times a string is repeated in another string
- Use `count()` function

```
my_string = 'Count, the number of spaces'
print("my_string.count(' '):", my_string.count(' '))
print("my_string.count('a'):", my_string.count('a'))
```

```
my_string.count(' '): 4
my_string.count('a'): 1
```

Replacing strings

- One string can replace a substring in another string in Python String
- Use `replace()` function

```
welcome_message = 'Hello World!'
print(welcome_message.replace("Hello", "Goodbye"))
```

```
Goodbye World!
```

Finding substrings

- To find out if one string is a substring of another string using the `find()` function
  - this method takes a second string as a parameter and checks to see if that string is in the string receiving the `find()` function
    - `string.find(string_to_find)`

```
print('Edward Alun Rawlings'.find('Alun'))
```

```
7
```

  - this prints out the value 7
    - index of the first letter of the substring 'Alun' note strings are indexed from zero
  - return -1 if the string is not present

```
print('Edward John Rawlings'.find('Alun'))
```

```
-1
```

Comparing strings

- To compare one string with another you can user the ‘==‘ equality and ‘!=‘ not equals operators
  - return either `True` or `False` indicating whether the strings are equal or not

```
print('James' == 'James')
print('James' == 'John')
print('James' != 'John')
```

```
True
False
True
```

  - Should note that strings in Python are case sensitive, so string ‘James’ does not equal the string ‘james’

Other string operations

- There are in fact very many different operations available for string
  - including checking that a string starts or ends with another string,
  - that is it upper or lower case,
  - to replace part of a string with another string,
  - convert strings to upper, lower, or title case, etc.

Other string operations

```
some_string = 'Hello World'
print('Testing a String')
print('-' * 20)
print('some_string', some_string)
print("some_string.startswith('H')",
some_string.startswith('H'))
print("some_string.startswith('h')",
some_string.startswith('h'))
print("some_string.endswith('d')", some_string.endswith('d'))
print('some_string.istitle()', some_string.istitle())
print('some_string.isupper()', some_string.isupper())
print('some_string.islower()', some_string.islower())
print('some_string.isalpha()', some_string.isalpha())
print('String conversions')
print('-' * 20)
print('some_string.upper()', some_string.upper())
print('some_string.lower()', some_string.lower())
print('some_string.title()', some_string.title())
print('some_string.swapcase()', some_string.swapcase())
print('String leading, trailing spaces', " xyz ".strip())
```

```
Testing a String
--------------------
some_string Hello World
some_string.startswith('H') True
some_string.startswith('h') False
some_string.endswith('d') True
some_string.istitle() True
some_string.isupper() False
some_string.islower() False
some_string.isalpha() False
String conversions
--------------------
some_string.upper() HELLO WORLD
some_string.lower() hello world
some_string.title() Hello World
some_string.swapcase() hELLO wORLD
String leading, trailing spaces xyz
```

Hints on strings

- Python strings are case sensitive
  - in Python, the string 'l' is not the same as the string 'L';
    - one contains the lower-case letter 'l' and one the upper-case letter 'L'
  - If case sensitively does not matter to you then you should convert any strings you want to compare into a common case before doing any testing

- Function/method names
  - be very careful with capitalization of function/method names;
    - `isupper()`, not `isUpper()`

Hints on strings

- Function/method invocations
  - be careful of always including the round brackets when you call a function or method;
    - event if it takes no parameters/arguments
  - There is a significant difference between `isupper` and `isupper()`

```
some_string = 'Heelo World'
print(some_string.isupper)
print(some_string.isupper())
```

```
<built-in method isupper of str object at 0x000002A6DCDA0EB0>
False
```

String formatting

- Python provides a sophisticated formatting system for strings that can be useful for printing information out or logging information from a program
- A special string known as `format` string that acts as a pattern defining how the final string will be laid out

```
format_string = 'Hello {}!'
print(format_string.format('Phoebe'))
```

```
Hello Phoebe!
```

- Can have any number of placeholders that must be populated

```
name = "Adam"
age = 20
print("{} is {} years old".format(name, age))
```

```
Adam is 20 years old
```

String formatting

- By default the value are bound to the placeholders based on the order that they are provided to the `format()` function
  - however, this can be overridden b providing an index to the placeholder to tell it which value should be bound

```
print("Hello {1} {0}, you got {2}".format('Smith', 'Carol', 75))
```

```
Hello Carol Smith, you got 75
```

  - alternative approach is to use named value for the placeholder

```
format_string = "{artist} sang {song} in {year}"
print(format_string.format(artist='Paloma Faith', song='Guilty', year=2017))
```

```
Paloma Faith sang Guilty in 2017
```

String formatting

- To indicate alignment and width within the format string
  - if you wish to indicate a width to be lf for a placeholder whatever the actual value supplied, use ‘:’ followed by the width to use
  - ex) to specify a gap of 25 characters which can be filled with a substituted value:

```
print('|{:25}|'.format('25 characters width'))
```

```
|25 characters width      |
```

String formatting

- To indicate alignment and width within the format string (cont'd)
  - within this gap you can also indicate an alignment where:
  - < indicates left alignment (default)
  - > indicate right alignment
  - ^ indicate centered

```
print('|{:<25}|'.format('left aligned')) # The default
print('|{:>25}|'.format('right aligned'))
print('|{:^25}|'.format('centered'))
```

```
|left aligned              |
|             right aligned|
|         centered         |
```

String formatting

- Integer alignment `{:[fill][align][width]d}`
    - `fill` is the character you want to use for filling (optional)
    - `align` is the alignment indicator (< for left, > for right, ^ for center)
    - `width` is the total width of the formatted string
    - `d` specifies that the argument is an integer

```
number = 123
print("{:>10d}".format(number))
print("{:^10d}".format(number))
print("{:0>10d}".format(number))
print("{:_<10d}".format(number))
print("{:.^10d}".format(number))
```

```
       123
   123
0000000123
123_______
...123....
```

String formatting

- Floating point alignment `{:[width].[precision]f}`
  - `width` specifies the total field width (including decimal point and digits)
  - `precision` specifies the number of digits after the decimal point

```
number = 123.45678
print("Basic formatting: {:.2f}".format(number))
print("Width 10, right-aligned: {:10.2f}".format(number))
print("Width 10, left-aligned: {:<10.2f}".format(number))
print("Width 10, centered: {:^10.2f}".format(number))
print("Width 10, zero-filled: {:010.2f}".format(number))
```

```
Basic formatting: 123.46
Width 10, right-aligned:     123.46
Width 10, left-aligned: 123.46
Width 10, centered:   123.46
Width 10, zero-filled: 0000123.46
```

String formatting

- Another formatting option
  - to indicate that a number should be formatted with separators (such as comma) to indicate thousands

```
print('{:,}'.format(1234567890))
print('{:,}'.format(1234567890.0))
```

```
1,234,567,890
1,234,567,890.0
```

String formatting

- Prefix f-string option in print() function
  - similar with `format()` function
  - provide a concise and readable way to embed expressions inside string literals (Python 3.6 or later)
  - allow with ‘f’ or ‘F’

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

```
My name is Alice and I am 30 years old.
```

String formatting

- % operator in print() function
  - similar with `format()` function
  - but, % operator is an older way of formatting strings in Python

```
print("Name: %s, Age: %d" % ("Alice", 30))
```

```
Name: Alice, Age: 30
```

  - %type
    - %d, %s, %c, etc.
    - C/C++ style print formatting

In class practice

- P02-02 다음 변수들을 아래 결과 처럼 예쁘게 출력하는 프로그램을 작성하라

```
a_id, a_name, a_major, a_income = 101, "Alice Smith", "Software Engineer", 75000.90
b_id, b_name, b_major, b_income = 102, "Bob Johnson", "Project Manager", 85000.50
c_id, c_name, c_major, c_income = 103, "Charlie Lee", "Data Analyst", 65000.00
d_id, d_name, d_major, d_income = 104, "David Wilson", "Intern", 32000.00

''' CODE HERE '''
```

```
ID    Name             Job Title              Salary
----  --------------   -----------------    ---------
 101 Alice Smith      Software Engineer    75,000.90
 102 Bob Johnson      Project Manager      85,000.50
 103 Charlie Lee      Data Analyst         65,000.00
 104 David Wilson     Intern               32,000.00
```

4. Numbers and Booleans

Note: Primary ways data represented in computers

- Binary numbers (or binary codes)
  - the most basic form of data representation
  - all data is represented as sequences of bits (0s or 1s)
  - numbers, characters, and even executable instructions can be encoded in binary

- Hexadecimal numbers
  - a more compact form of binary representation
  - four bits are represented by a single hexadecimal digit (0-9 and A-F)

- ASCII code (The American Standard Code for Information Interchange)
  - a character encoding standard used to represent text in computer

Note: Bit and byte

- Bit
  - the smallest unit of data in computer for a single binary value; either 0 or 1
  - can represent a range of different meanings
    - e.g., 1/0, on/off, true/false, or any other two-state system

Bit Width	Unsigned Range	Signed Range
8-bit	0 to $2^8 - 1$	-2^7 to $2^7 - 1$
16-bit	0 to $2^{16} - 1$	-2^{15} to $2^{15} - 1$
32-bit	0 to $2^{32} - 1$	-2^{31} to $2^{31} - 1$
64-bit	0 to $2^{64} - 1$	-2^{63} to $2^{63} - 1$

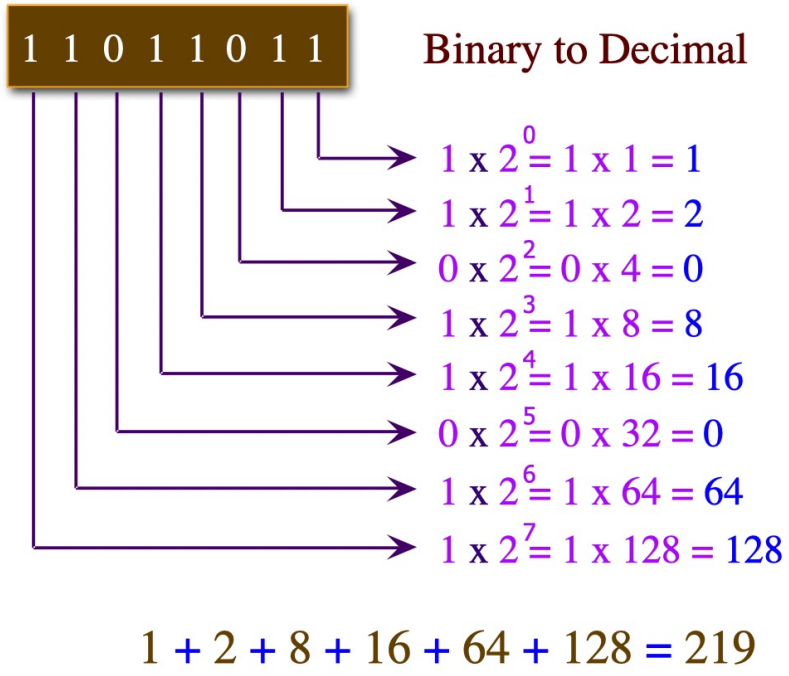
- Byte
  - a unit of digital information that most commonly consists of 8 bits
  - can represent 256 different values (from 0 to 255 in decimal)

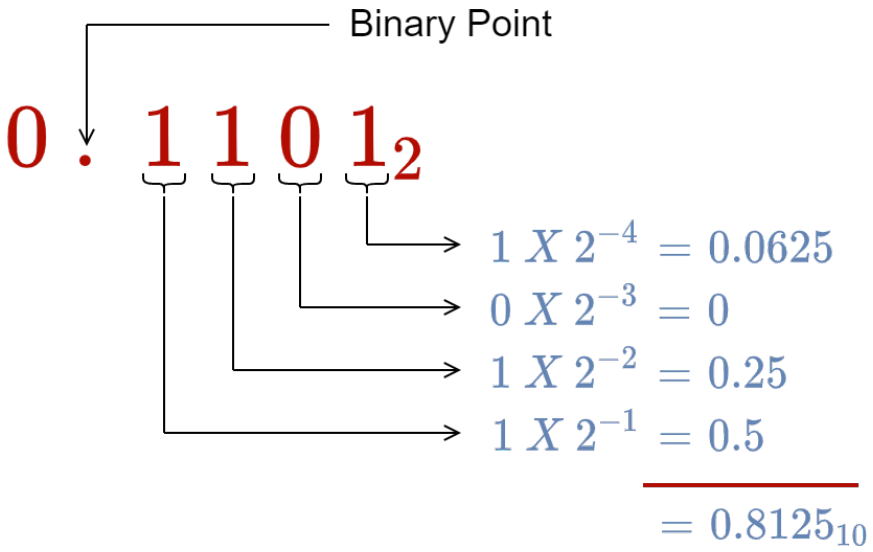
Note: Number system conversion

- Binary-decimal conversion

Decimal (BASE 10)	0	1	2	3	4	5	8	10
Binary (BASE 2)	0	1	10	11	100	101	1000	1010

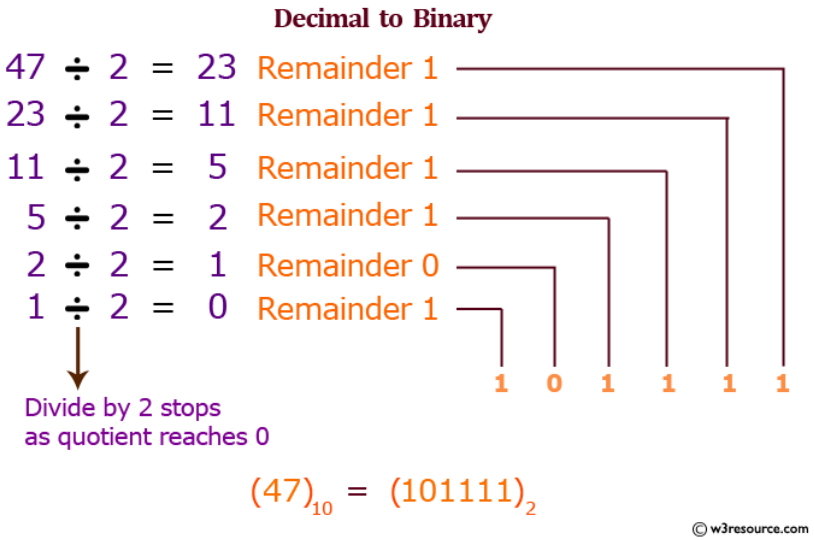
- binary to decimal conversion





Note: Number system conversion

- Binary-decimal conversion
  - decimal to binary conversion
    - integer part: divide this number repeatedly by 2 until the quotient becomes 0
    - fractional part: multiply the fractional part repeatedly by 2 until it becomes 0
    - example: 47.375 (decimal) to binary conversion



- fractional part: 0.375

  0.375 x 2 = 0.750

  0.750 x 2 = 1.500

  0.500 x 2 = 1.000

➔ 101111.011 (binary)

Note: Number system conversion

- Other number systems
  - Octal numbers: 0~7
  - Hexadecimal number: 0~9, A~F
  - How to convert
    - hexadecimal to decimal number
      - e.g., AB1 (hexa) ➔ ? (decimal)
    - octal to binary number
      - e.g., 1071 (octal) ➔ ? (binary)

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Note: Number system conversion

- e.g., AB1 (hexa) ➔ ? (decimal)

$A * 16^2 = 10 * 256 = 2560$
$B * 16^1 = 11 * 16 = 176$
$1 * 16^0 = 1 * 1 = 1$
$2560 + 176 + 1 = 2737$

- e.g., 1071 (octal) ➔ ? (binary)

1 → $(001)_2$
0 → $(000)_2$
7 → $(111)_2$
1 → $(001)_2$
$(001\ 000\ 111\ 001)_2$

Types of numbers

- Three types used to represent numbers in Python
  - integers, floating point numbers, complex numbers

- Why have different ways of representing numbers?
  - human
    - can easily work with the number 4 and 4.0
    - don't need completely different approaches to writing them
  - computer
    - comes down to efficiency in terms of both the amount of memory needed to represent a number
    - integers are simpler to work with and can take up less memory than real numbers

Integers

- All integer values, no matter how big or small are represented by the integer type in Python

```
x = 1
print(x)
print(type(x))
x = 10000000000000000000000000000000000000000000000000000000000000001
print(x)
print(type(x))
```

```
1
<class 'int’>
10000000000000000000000000000000000000000000000000000000000000001
<class 'int'>
```

Converting into integers

- To convert another type into an integer using the `int()` function

```
total = int(1.0)
total = int(1.234)
total = int('100')
```

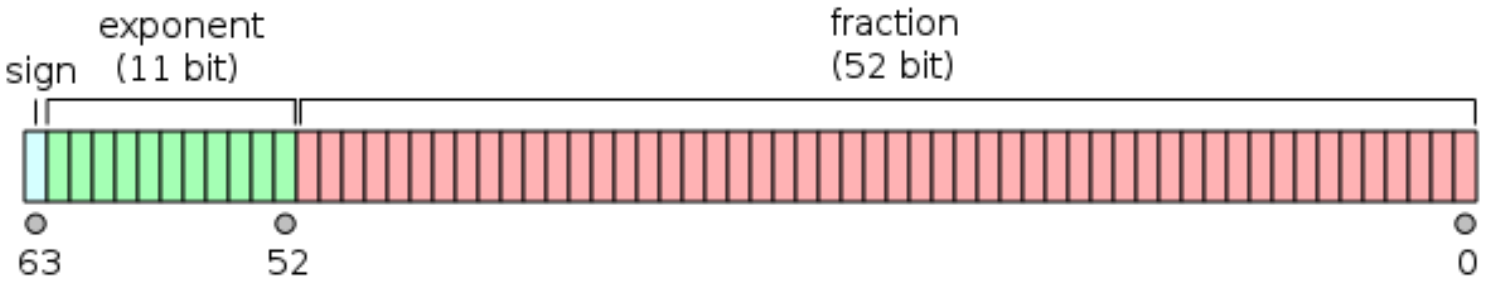
- It is useful when used with the `input()` function

```
age = input('Please enter your age:')
print(type(age))
print(age)

age = int(input('Please enter your age:'))
print(type(age))
print(age)
```

```
Please enter your age:12345
<class 'str'>
12345
Please enter your age:12345
<class 'int'>
12345
```

Floating point numbers

- Real numbers are represented as floating point numbers (or floats)
  - an integer part + a fractional part (the bit after the decimal point)
  - computer can best work with integers (only 1s and 0s)
  - need a way to represent a floating point or real number; total 64 bits

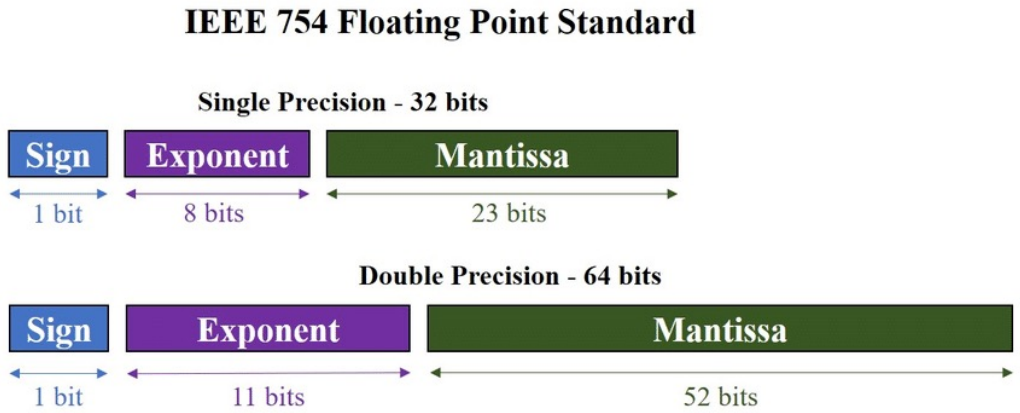


    - by IEEE 754 double-precision binary floating point number format

Floating point numbers

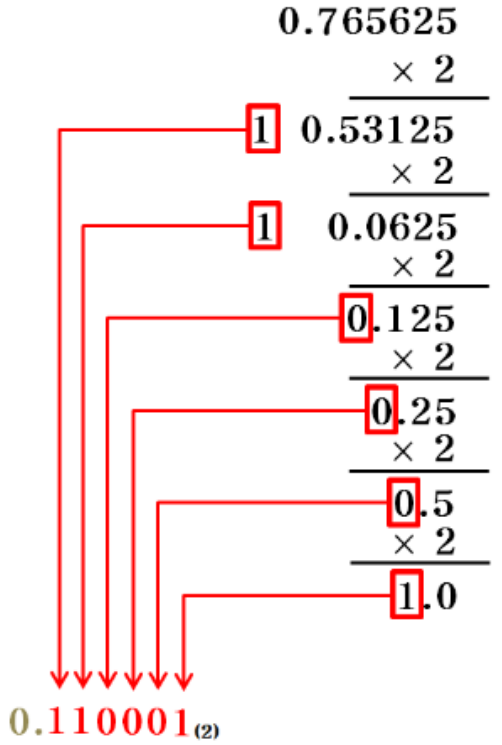
- Real number = integer (decimal) part + fractional (mantissa) part
  - 23.1519 = 23 + 0.1519

- Floating-point (부동소수점) by IEEE 754 floating point standard
  - a method to represent real numbers in computer
  - data types in Java: 'float' and 'double'
  - why are the real numbers represented by floating-point?
    - floating-point numbers cannot precisely represent all real numbers
      - ➔ precision and rounding errors
      - ex) 0.1 (decimal) ➔ 0.0011001100110011..... (binary)



Floating point numbers

- Example for 11.765625 (decimal) (only float type)
  - 1) representation in binary ➔ 1011.110001 (binary)
    - integer part: 11 (decimal) ➔ 1011 (binary)
    - fractional part: 0.765625 (decimal) ➔ 0.110001 (binary)
  - 2) normalize the binary number
    - 1011.110001 ➔ 1.01110001 * 2^3
  - 3) determine the exponent
    - the exponent is 3 ➔ 130 (=127 + 3) = 10000010
      - 127 is is the bias for float type; double is 1023
  - 4) encode the fraction
    - 01110001 (ignoring the leading 1)



exponent (8bit)

01000001001110001000000000000000

sign (1bit) mantissa (23bit)

Floating point numbers

- Example for 0.15625 (decimal)
  - 1) representation in binary ➔ 0.00101 (binary)
  - 2) normalize the binary number
    - 0.00101 ➔ 1.01 * 2^(-3)
  - 3) determine the exponent
    - the exponent is -3 ➔ 124 (=127 - 3) = 01111100
  - 4) encode the fraction
    - 01 (ignoring the leading 1)

Floating point numbers

- How to obtain the memory size for higher-level data type

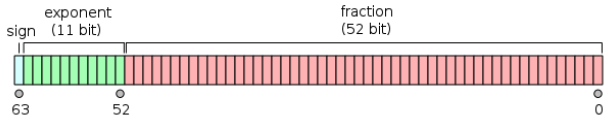
```
import sys

int_size = sys.getsizeof(100)
float_size = sys.getsizeof(101.1239930)
char_size = sys.getsizeof('100')

print(f"Size of an integer in bytes: {int_size}")
print(f"Size of a float in bytes: {float_size}")
print(f"Size of a character in bytes: {char_size}")
```

```
Size of an integer in bytes: 28
Size of a float in bytes: 24
Size of a character in bytes: 50
```

  - Python's data types are dynamically sized (primitive size + overhead size)
    - unlike Java, C, C++

Floating point numbers

- Why is different the result of `sys.getsizeof(101.1239930)` from size of floating point number format?
  - `sys.getsizeof(101.1239930)` = 24 bytes
  - Remember IEEE 754 double-precision binary floating point number format ➔ 8 bytes (64 bits)



  - ➔ the remaining bytes, 16 (24 – 8) bytes, for the Python object overhead and memory alignment
    - object overhead: some additional information (reference count, type, other bookkeeping info.)
    - memory alignment: padding to align the objects' data in memory for faster access

Floating point numbers

- Real number, or floating point number

```
exchange_rate = 1.83
print(exchange_rate)
print(type(exchange_rate))
```

```
1.83
<class 'float'>
```

- Due to fractional representation, it cannot provide the “precise” value

```
a = 1.83 * 1.000000001
a == 1.83

a = 1.83 * 1.0000000000000001
a == 1.83
```

```
False
True
```

Floating point numbers

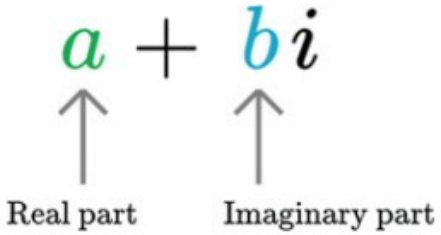
- To convert another type into a float using the `float()` function

```
int_value = 1
string_value = '1.5'
float_value = float(int_value)
print('int value as a float:', float_value)
print(type(float_value))
float_value = float(string_value)
print('string value as a float:', float_value)
print(type(float_value))
```

```
int value as a float: 1.0
<class 'float'>
string value as a float: 1.5
<class 'float'>
```

Complex numbers

- Defined by a real part and an imaginary part; a+bi




```
c1 = 2j
c2 = -5+3j
print('c1:', c1, ', c2:', c2)
print(type(c1), type(c2))
print(c1.real, c1.imag)
print(c2.real, c2.imag)
```

```
c1: 2j , c2: (-5+3j)
<class 'complex'> <class 'complex'>
0.0 2.0
-5.0 3.0
```

- the letter ‘j’ is used in Python to represent the imaginary part of the number

Quiz

- What is the output of the following code?

```
complex_number = 4 + 3j
print(abs(complex_number))
```

  - abs()
    - make absolute number

  - a) 4
  - b) 5
  - c) 5.0
  - d) 7

Boolean values

- Python supports another type called Boolean;
  - only be one of `True` or `False` (and nothing else)
    - with capital T and F; not ‘true’ and ‘false’

```
all_ok = True
print(all_ok)
all_ok = False
print(all_ok)
print(type(all_ok))
```

```
True
False
<class 'bool'>
```

Boolean values

- Boolean type is actually a sub type of integer (but with only the values `True` and `False`)
  - easy to translate between the two, using function `int()` and `bool()`

```
print(int(True))
print(int(False))
print(bool(1))
print(bool(0))
```

```
1
0
True
False
```

- Can also convert strings into Booleans as long as the strings contain either `True` or `False`

Boolean values

- Can also determine `True` or `False` by whether there is a value in the string

```
status = bool(input('OK to proceed: '))
print(status, type(status))

status = bool(input('OK to proceed: '))
print(status, type(status))

status = bool(input('OK to proceed: '))
print(status, type(status))
```

```
OK to proceed: True
True  <class 'bool’>

OK to proceed: False
True  <class 'bool'>

OK to proceed:
False <class 'bool'>
```

Quiz

- What is the output of the following code?

```
print(1+True)
```

  - a) Error
  - b) 1
  - c) 2
  - d) True

Quiz

- What is the output of the following code?

```
print(max(min(False, False), 1, True))
```

  - a) Error
  - b) 1
  - c) True
  - d) 0

End of slide