TEMASEK
JUNIOR COLLEGE

Temasek Junior College
2022 JC1 H2 Computing
Practical 2 – Working with Variables and Identifiers

Session Objectives

By the end of this session, you will learn:

(i) what a variable is and its importance in the programming context.
(ii) how to assign a value to a variable.
(iii) how variables are associated to the memory.
(iv) how to inspect the value of a variable.
(v) the creation of valid and meaningful identifiers in Python.
(vi) the reserved words in Python.
(vii) the conventions for creating identifiers for variables in Python.

§2.1 Variables

§2.1.1 What is a Variable?

In programming, **variables** are identifiers (names) associated with a particular memory location used to store data. By using a variable, a programmer can store, retrieve and manipulate data without having to know what the actual data is.

Variables are fundamental to programming for two key reasons:

(1) **Keep values accessible** e.g. you can assign the result of an operation to a variable so that the operation need not be repeated each time you need to use the result.

(2) **Give values context** e.g assigning the value 250 to a meaningful identifier e.g. `daily_salary`, makes it clear that the salary rate is 250 dollars per day.

§2.1.2 Assigning Values to Variables

An **operator** is a symbol that performs an operation on one or more values e.g the addition operator + takes two numbers, one left of + and one right of +, then adds them together.

Values are assigned to variables using the **assignment operator =**. It takes the value (not necessarily a number) to the right of the operator and assigns it to the identifier on the left. For example, the code `x = 5` means assigning the value `5` to the variable `x`.

It is important to note that while the = symbol refers to "equals to" in Mathematics, it has a different meaning in Python.

Question

What then is operator that works as "equals to" in Python?

Answer

==

Exercise 1

Type and execute the following code in the REPL:

```
greeting = 'Hello, World'
print(greeting)
```

Observe what happens.

Let us now attempt to understand what you have done in **Exercise 1**:

(1) In the first line, you assigned the value `'Hello, World'` to the variable with identifier greeting using the = operator.

(2) In the second line, `print(greeting)` makes Python look for the identifier greeting. It finds that greeting has been assigned the value `'Hello, World'`. It then replaces greeting with the value `"Hello, World"`. The `print()` function is then invoked and `'Hello World'` is printed as the result.

Question

What happens if you did not execute the line `greeting = 'Hello, World'` before executing the line `print(greeting)`? Why?

Answer

NameError: name 'greeting' is not defined

§2.1.3 Variables and its Association to Memory

A Python variable is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object using the variable. The data itself is still contained within the object.

Exercise 2

Type and execute the following code in the REPL:

```
x = 500
y = x
print(x)
print(y)
id(x)
id(y)
```

What do you observe? Why?

Answer

- A variable `x` is created and a particular location in the memory is used to hold a newly created Python object with value 500. A unique "identification number" represents this particular location in memory. Variable `x` points to this location, hence it follows the "identification number".
- A variable `y` is then created. The variable `x` is assigned to `y`. This will imply that `y` also points to the location in the memory where the Python object with value 500 is held. Variable `y` will thus follow the same "identification number".
- Hence `id(x)` and `id(y)` are the same and both `x` and `y` have the same value.

Exercise 3

Type and execute the following code in the REPL:

```
x = 500
y = 500
print(x)
print(y)
id(x)
id(y)
```

Is `id(x)` the same as `id(y)`? Why?

Answer

No.

Each assignment of a value to a variable triggers the creation of a new Python object. `x = 500` and `y = 500` triggered the creation of two Python objects with value 500, held in two different locations in the memory. Hence there are two different but unique "identification numbers". Since `x` and `y` are pointing to different memory locations, they will follow different "identification numbers" even though the value of `x` and that of `y` are the same.

Exercise 4

Type and execute the following code in REPL:

```
x = 50
y = 50
print(x)
print(y)
id(x)
id(y)
```

Is `id(x)` the same as `id(y)`? Why?

Answer

Yes.

To optimise processing speed, Python pre-creates objects that have integer values in the interval [–5, 256] at start-up and then reuses them during program execution. When you assign an integer value in this interval to separate variables, these variables will point to the same object.

Besides integer values in the interval [–5, 256], Python also does the same for short strings.

Exercise 5

(i) Type and execute the following code in REPL:

```
x = 'Hi'
y = 'Hi'
id(x)
id(y)
```

Is `id(x)` the same as `id(y)`? Yes

(ii) Type and execute the following code in REPL:

```
x = 'Hi, I am from CG 03/21.'
y = 'Hi, I am from CG 03/21.'
id(x)
id(y)
```

Is `id(x)` the same as `id(y)`? No

§2.1.4 Variable Inspection on the REPL

Exercise 6

Type and execute the following code in the REPL:

```
greeting = 'Hello, World'
greeting
```

Observe what happens.

When you press Enter after typing greeting a second time, the string literal 'Hello, World' assigned to greeting is displayed even though you didn't use the `print()` function. This is called **variable inspection**.

Exercise 7

Add and execute the following line to the code in Exercise 7:

```
print(greeting)
```

Can you spot the difference between the results displayed by using the `print()` function and the results displayed by variable inspection?

- When the identifier greeting is executed without using the `print()` function, the value assigned to greeting, which is the string literal 'Hello, World' with the quotation marks is displayed as the results.
- With the `print()` function, Python evaluates the line as instruction to print the text enclosed within the quotation marks. Hence Hello, World without quotation marks is printed.

Sometimes, the same result may be displayed for variable inspection and the `print()` function.

Exercise 8

Type and execute the following code in the REPL:

```
x = 2
x
print(x)
```

What do you notice? There is no difference.

In **Exercise 8**, you assigned the number 2 to the variable x. Both `print(x)` and the variable inspection of x results in 2 being displayed. This is because 2 is a number and not text (hence no quotation marks were used in the first place).

Nevertheless, in most cases, performing a variable inspection gives you more useful information than using the `print()` function.

Exercise 9

(i) Type and execute the following code in the REPL:

```
x = 2
y = '2'
print(x)
print(y)
```

(ii) Add and execute the following lines of code:

```
x
y
```

(iii) What do you notice? x has no quote while y has quotes.

In **Exercise 9**, you assigned the integer 2 to x and the string '2' to y. When the print function is used, it is not possible differentiate which result is an integer and which is a string literal since they are both identical.

Instead, a variable inspection of x will reveal that its value assigned is the integer 2 as displayed and a variable inspection of y will reveal that its value assigned is the string literal '2' as displayed (with quotation marks).

The key takeaway here is that the `print()` function displays a readable representation of a variable's value, while variable inspection displays the value as how it was coded.

Exercise 10

In the **File Editor**, code and execute the following:

```
greeting = 'Hello World'
greeting
```

Does the programme display any results? What can you say about variable inspection?

Answer

Variable inspection only works in the REPL.

In **Exercise 10**, the code was executed successfully. No errors were reported. However, no results were displayed.

§2.2 Identifiers

§2.2.1 Case Sensitivity in Using Identifiers

Identifiers are case sensitive e.g. the identifiers `greeting` and `Greeting` are different.

Exercise 11
Type and execute the following code in the REPL:

```
greeting = 'Hello, World'
print(Greeting)
```

Observe what happens.

In **Exercise 11**, a `NameError` occurs. This is because the variable `Greeting` has no value assigned to it. The first line of code assigned the value Hello, World to greeting and not `Greeting`.

§2.2.2 Rules for Valid Identifiers

There are a few rules that you must follow when creating valid identifiers.

- Identifiers may contain:
  - ✓ uppercase and lowercase English letters (A – Z, a – z),
  - ✓ underscores (_), and
  - ✓ digits (0 – 9)
- An identifier however **CANNOT** begin with a digit.
- An identifier also **CANNOT** contain special characters such as !, @, #, $, %.

Exercise 12
Determine if the following identifiers are valid.
If the identifier is invalid, state the reason.

(a) string1	valid
(b) 9lives	Invalid, SyntaxError, Invalid decimal literal
(c) 2beOrNot2Be	Invalid, SyntaxError, Invalid decimal literal
(d) _a1ph@	Invalid Syntax
(e) list_of_names	valid
(f) 99balloons	Invalid, SyntaxError, Invalid decimal literal

In addition to English letters and digits, Python identifiers may contain many different valid **Unicode** characters.

Unicode is a standard for digitally representing characters used in most of the world's writing systems. As such, identifiers can contain letters from non-English alphabets, such as decorated letters like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it will be wise to avoid them if you want your code to be readable in all geographical regions.

As a rule of thumb, just use uppercase and lowercase English letters, underscores and digits.

We shall learn more about Unicode in our course of study.

§2.2.3 Conventions for Writing Identifiers for Variables in Python

In many programming languages, it is common to write identifiers for variables in the mixedCase. In this system, you capitalize the first letter of every word except the first and leave all other letters in lowercase e.g. numStudents and listOfNames.

In Python, identifiers for variables are common written in lower_case_with_underscores. In this system, you leave every letter in lowercase and separate each word with an underscore e.g. num_students and list_of_names. This practice is documented in PEP 8, which is widely regarded as the official style guide for coding Python.

Following the standards outlined in PEP 8 ensures that your Python code is readable by most Python programmers. This makes sharing code and collaborating with other people easier for everyone involved.

Despite so, there is really no rule mandating that you need to write identifiers for variables in lower_case_with_underscores.

It is important to note that the use of lower_case_with_underscores **applies for variables**. We will learn other naming conventions in our course of study.

§2.2.4 Reserved Words

Now that we know how to create an identifier, we need to familiarise ourselves with a list of words that should not be chosen as identifiers.

These words are reserved words, or keywords of the Python language, and cannot be used as ordinary identifiers. They are case sensitive and must be typed exactly as seen.

The following is a list of words that should not be chosen as identifiers in Python 3.6.4.

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

Exercise 13
Type and execute the following code in the REPL:

```
import keyword
keyword.kwlist
```

What do the two lines of code produce?

Answer
List of reserved words.

§2.2.5 Meaningful Identifiers

An identifier that is valid may not necessarily mean that it is a good identifier. **Meaningful identifiers** are essential to:

- show explicitly the purpose for which the identifier is used for.
- enhance readability of the code to programmers and users.
- ease the effort required to maintain and enhance the code.

Consider the following:

```
s = 3600
```

The value 3600 is assigned to the identifier `s`. However, the identifier `s` does not give any information on what 3600 might refer to.

Using a full word as an identifier can make it easier to the comprehend the meaning of 3600:

```
seconds = 3600
```

The identifier seconds is better than s since we now know that 3600 is a measure of time in seconds. However, it still does not convey the full meaning of 3600.

Is it the number of seconds it takes for a process to finish, or is it the length of a movie?
A more descriptive identifier is needed to make the context clear.

```
seconds_per_hour = 3600
```

There is no doubt now that 3600 is the number of seconds in an hour.
While the identifier `seconds_per_hour` takes longer to type than both s and seconds, the gain in clarity is significant.

Writing descriptive identifiers often requires using multiple words. It is thus possible to have long variable names. Nevertheless, you should avoid using excessively long names. A good rule of thumb is to limit variable names to three or four words maximum.

Tutorial

Complete the following questions and submit your file for Question 1 on Google Classroom.

1) Write a Python program that fulfils the following requirements:

- Assigns a string literal to a variable with a meaningful identifier.
- Prints the value of the variable when executed.

2) We have used the function `id()` in this practical session. What does `id()` give you? How does it work?

Hint: Python has a `help()` function. Explore how the `help()` function works and use it to aid you in answering this question