Introduction to Python: Variables, Strings, and Numbers

Let us try the famous "hello world" Note that we are using "***Python 3***"

Created by John C.S. Lui, May 24, 2018

Important note: *If you want to use and modify this notebook file, please acknowledge the author.*

In [ ]:
```
print ("Hello world, this is great, my friend")
print ('this is another string.')
```

Let see how we can create and assign variables

In [ ]:
```
message = "Hello to the wonderful world of Python,"    # assign a stri
print (message)
```

In [ ]:
```
message2 = "and welcome to CSCI2040."
print (message)
print (message2)
```

In [ ]:
```
print (message, message2)          # try to print both messages in one
```

Naming rules

1. Variables can only contain letters, numbers, and underscores. Variable names can start with a letter or an underscore, but can not start with a number.
2. Spaces are not allowed in variable names, so we use underscores instead of spaces. For example, use student_name instead of "student name".
3. You cannot use Python keywords as variable names.
4. Variable names should be descriptive, without being too long. For example mc_wheels is better than just "wheels", and number_of_wheels_on_a_motorycle.
5. Be careful about using the lowercase letter l and the uppercase letter O in places where they could be confused with the numbers 1 and 0.

Exercises

- Create a variable, assign it with some value and print it out
- Reassign the variable of some new value and print it out

Let's learn something about "*strings*"

In [ ]:
```
my_string1 = "a string with a double quote"
my_string2 = 'a string with a single quote'
print ("STR1=", my_string1)
print ('STR2=', my_string2)   # let's see the output
```

In [ ]:
```
# what about a string which contains a quote?
quote = "Martin Luther King Jr. said, 'Free at last, free at last, thar
print (quote)
```

Changing cases for the string

In [ ]:
```
first_name = 'john'

print(first_name)           # just print the string
print(first_name.title())   # capitalize the first letter
print(first_name.upper())   # capitalize the whole string
```

In [ ]:
```
last_name = "Lui"

print(last_name.lower())    # Let's be-little John
```

String, turns out, is a data type with **MANY** built-in functions. Demonstrate in class how to look up from Python standard library, e.g., :

The Python 3 Standard Library -> 4.7. Text Sequence Type — str

Then search for, let say, 'upper'.

How can we combine strings? Use cancatenation.

In [ ]:
```
print ("John's first name is =", first_name)
print ("John's last name is  =", last_name)

full_name = first_name + last_name
print ("John's name is = ", full_name)
```

In [ ]:
```
full_name1 = first_name.title() + " " + last_name   # capaitalize the
print ("John's full name is = ", full_name1)
```

In [ ]:
```
my_message = full_name1 + ', ' + "is a real jerk !!!!!"
print (my_message)
```

Adding control characters into strings

For example, how can we add *tab* or *line feed* into our strings?

In [ ]:
```
print("John is", "a jerk")    # automatically add one space
print("John is\t", "a jerk")  # adding a tab
print("John is\t\t\t", "a jerk!!!!")    # adding 3 tabs
print('But Mr. Cy Leung is even a bigger jerk !!!!!')
```

In [ ]:
```
print("John is \n", "a jerk.")     # a new line
print("\nJohn is\n", "\t\treally a jerk.") # a new line and some tabs
print('So my advice is: \t\t\t Immediately drop the course CSCI2040 !!
```

Stripping white spaces

When people input something, they may inadvertently add more "white spaces" than they intended to. What should we do?

In [ ]:
```
name = "  CUHK  "      # a string with white spaces, before and after
print("stripping left of the string, name="+ name.lstrip())   # stripp
print("stripping right of the string, name="+ name.rstrip())   # strip
print("stripping both sides of the string, name="+ name.strip())   # s
```

In [ ]:
```
# a better to visualize the above functions
name = "  CUHK  "      # a string with white spaces, before and after
print("stripping left of the string, name=" + " ***" + name.lstrip() +
print("stripping right of the string, name=" +  " ***" + name.rstrip()
print("stripping both sides of the string, name=" +  " ***" + name.str
```

arguments to methods

- There are many built-in methods to strings, as well as other built-in data-types.
- For each built-in method, it may take in some arguments
- Let's look at the documentation for lstrip() and see what it can drop

In [ ]:
```
# Instead of using the default, demonstrate passing argument to the bu

name = 'CUHK'
print ("I love", name)
name_1 = name.lstrip('UC')     # left strip the argument 'UC'

name_2 = name_1 + "U"
print ('I hate ' + name_2 + ", it is the worst univeristy in", name_1 +
```

Exercise

Create a similar output as:

Oscar Wilde, an Irish poet, once said,
"When I was young,
    I thought that money was the most important thing in life;
        now that I am old,
            I know that it is."

Exercise

- Store your first name in a variable, but include two different kinds of whitespace on each side of your name.
- Print your name as it is stored.
- Print your name with whitespace stripped from the left side, then from the right side, then from both sides.

Numbers and numerics

Integers

In [ ]:
```
print (1+2)   # addition
print (3-2)   # subtraction
print (5*2)   # multiplication
print (8/4)   # division
print (2**4)  # exponentiation
```

In [ ]:
```
print (1+2.)   # addition
print (3.-2)   # subtraction
print (5*2.)   # multiplication
print (8/4)    # division
print (2.0**4)  # exponentiation
```

In [ ]:
```
# Use parenthesis to modify the standard order of operations.
print (2+3*4)
print ((2+3)*4)
```

Floating points

In [ ]:
```
print (0.1+0.1)
print (0.1+0.2)
```

Division in Python 2.7 and Python 3.3

- In Python 2.7; print 3/2 will give you 1
- In Python 3.3, proint 3/2 will give you 1.5

Some common operators: +, -, , *, /, //, %

Let's illustrate some common operators when they are applied to different data types

In [ ]:
```
print(3+5)                  # adding two integers
print(3. + 5)               # type conversion
print('Aaa' + 'Bbbbbb' +'CCC')      # cancatenation
```

In [ ]:
```
print(-5.2)           # gives a negative number
print(5-3)
print(2*3)
print('Lu'*2)         # generate character strings
print(3**4)           # power
print (3.0**4.)
print(5/3)
print(5/3.)
print(4//3)            # gives floor
print(5.0//3.)
print (17 % 3)         # gives remainder
print (17. % 3)
```

In [ ]:
```
stuff = 'hello world'
type(stuff)    # see what type is this variable
num1 = 12
num2 = 12.
print ('stuff is of type: ', type(stuff))
print ('num1 is of type: ', type(num1))
print ('num2 is of type: ', type(num2))
```

In [ ]:
```
dir(stuff)     # show all built-in functions of tyis type
```

In [ ]:
```
help(stuff.capitalize)    # show the meaning of the capitalize functio
```

Comments

This line is a comment.
Note that # can also be put after a computational statement

Good pratices for comments

- It is short and to the point, but a complete thought.
- It explains your thinking, so that when you return to the code later you will understand how you were approaching the problem.
- It explains your thinking, so that others who work with your code will understand your overall approach.
- It explains particularly difficult sections of code in detail.

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

In [ ]: