CS 121
Advanced Computer Programming

Ms. Glydel Ann Reyes

Introduction to Python

Python

- Python was developed in the early 1990's by Guido van Rossum, then at CWI in Amsterdam, and currently at CNRI in Virginia.
- Python is a high-level scripting language which can be used for a wide variety of text processing, system administration and internet-related tasks.
- Unlike many similar languages, it's core language is very small and easy to master, while allowing the addition of modules to perform a virtually limitless variety of tasks.
- Python is a true object-oriented language, and is available on a wide variety of platforms.
- Open source general-purpose language.
- Great interactive environment.

The Basic Elements of Python

LITERALS

- In the following example, the parameter values passed to the print function are all technically called ***literals***

```
>>> print("Hello")
Hello
>>> print("Programming is fun!")
Programming is fun!
>>> print(3)
3
>>> print(2.3)
2.3
```

SIMPLE ASSIGNMENT STATEMENT

- A literal is used to indicate a specific value, which can be *assigned* to a *variable*

x is a variable and 2 is its value

```
>>> x = 2
>>> print(x)
2
>>> x = 2.3
>>> print(x)
2.3
```

- Python assignment statements are actually slightly different from the “variable as a box” model
- In Python, values may end up anywhere in memory, and variables are used to refer to them
- Interestingly, as a Python programmer you do not have to worry about computer memory getting filled up with old values when new values are assigned to variables
- Python will automatically clear old values out of memory in a process known as *garbage collection*

ASSIGNING INPUT

- So far, we have been using values specified by programmers and printed or assigned to variables
  - How can we let users (not programmers) input values?
  - In Python, input is accomplished via an assignment statement combined with a built-in function called *input*
- When Python encounters a call to *input*, it prints <prompt> (which is a string literal) then pauses and waits for the user to type some text and press the <Enter> key

<variable> = input(<prompt>)

- Here is a sample interaction with the Python interpreter:

```
>>> name = input("Enter your name: ")
Enter your name: Mohammad Hammoud
>>> name
'Mohammad Hammoud'
>>>
```

- Notice that whatever the user types is then stored as a string
  - What happens if the user inputs a number?

- Here is a sample interaction with the Python interpreter:

```
>>> number = input("Enter a number: ")
Enter a number: 3
>>> number
'3'
>>>
```

- How can we force an input number to be stored as a number and not as a string?
  - We can use the built-in ***eval*** function, which can be “wrapped around” the input function

Still a string!

- Here is a sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter a number: "))
Enter a number: 3
>>> number
3
>>>
```

Now an int
(no single quotes)!

- Here is a sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter a number: "))
Enter a number: 3.7
>>> number
3.7
>>>
```

And now a float
(no single quotes)!

- Here is another sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter an equation: "))
Enter an equation: 3 + 2
>>> number
5
>>>
```

The *eval* function will evaluate this formula and return a value, which is then assigned to the variable “number”

DATATYPE CONVERSION

- Besides, we can convert the string output of the *input* function into an integer or a float using the built-in ***int*** and ***float*** functions

```
>>> number = int(input("Enter a number: "))
Enter a number: 3
>>> number
3
>>>
```

An integer
(no single quotes)!

- We can also convert the string output of the *input* function into an integer or a float using the built-in ***int*** and ***float*** functions

```
>>> number = float(input("Enter a number: "))
Enter a number: 3.7
>>> number
3.7
>>>
```

A float
(no single quotes)!

- As a matter of fact, we can do various kinds of conversions between strings, integers and floats using the built-in ***int***, ***float***, and ***str*** functions

```
>>> x = 10
>>> float(x)
10.0
>>> str(x)
'10'
>>>
```

```
>>> y = "20"
>>> float(y)
20.0
>>> int(y)
20
>>>
```

integer ➔ float
integer ➔ string

string ➔ float
string ➔ integer

```
>>> z = 30.0
>>> int(z)
30
>>> str(z)
'30.0'
>>>
```

float → integer
float → string

SIMULTANEOUS ASSIGNMENT

- This form of assignment might seem strange at first, but it can prove remarkably useful (e.g., for swapping values)

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
>>>
```

- Suppose you have two variables x and y, and you want to swap their values (*i.e., you want the value stored in x to be in y and vice versa*)

```
>>> x = 2
>>> y = 3
>>> x = y
>>> y = x
>>> x
3
>>> y
3
```

CANNOT be done with *two* simple assignments

- Suppose you have two variables x and y, and you want to swap their values (*i.e., you want the value stored in x to be in y and vice versa*)

```
>>> x = 2
>>> y = 3
>>> temp = x
>>> x = y
>>> y = temp
>>> x
3
>>> y
2
>>>
```

Thus far, we have been using different *names* for variables. These names are technically called ***identifiers***

CAN be done with *three* simple assignments, but more efficiently with simultaneous assignment

IDENTIFIERS

- Python has some rules about how identifiers can be formed
  - Every identifier must begin with a letter or underscore, which may be followed by any sequence of letters, digits, or underscores

```
>>> x1 = 10
>>> x2 = 20
>>> y_effect = 1.5
>>> celsius = 32
>>> 2celsius
  File "<stdin>", line 1
    2celsius
           ^
SyntaxError: invalid syntax
```

- Python has some rules about how identifiers can be formed
  - Identifiers are *case-sensitive*

```
>>> x = 10
>>> X = 5.7
>>> print(x)
10
>>> print(X)
5.7
```

- Python has some rules about how identifiers can be formed
  - Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers

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	

- Python has some rules about how identifiers can be formed
  - Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers

```
>>> for = 4
File "<stdin>", line 1
        for = 4
            ^
SyntaxError: invalid syntax
```

- Python has some rules about how identifiers can be formed
  - Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers

```
>>> for = 4
File "<stdin>", line 1
        for = 4
            ^
SyntaxError: invalid syntax
```

Expressions

- You can produce new data (numeric or text) values in your program using *expressions*

```
>>> x = 2 + 3
>>> print(x)
5
>>> print(5 * 7)
35
>>> print("5" + "7")
57
```

- This is an expression that uses the *addition operator*
- This is another expression that uses the *multiplication operator*
- This is yet another expression that uses the *addition operator* but to *concatenate* (or glue)
  strings together

- You can produce new data (numeric or text) values in your program using *expressions*

```
>>> x = 6
>>> y = 2
>>> print(x - y)
4
>>> print(x/y)
3.0
>>> print(x//y)
3
```

```
>>> print(x*y)
12
>>> print(x**y)
36
>>> print(x%y)
0
>>> print(abs(-x))
6
```

Summary of Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Float Division
**	Exponentiation
abs()	Absolute Value
//	Integer Division
%	Remainder

BRANCHING PROGRAMS

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **Concatenate** strings

```
name = "ana"
greet = hi + name
greeting = hi + ""+ name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + ""+ name * 3
```

LOGIC OPERATORS ON bools

A and bare variable names (with Boolean values)
not a True if a is False False if a is True
a and b True if both are True
a or b True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

```
pset_time= 15
sleep_time= 8
print(sleep_time> pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```

CONTROL FLOW –BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

CONTROL FLOW: while LOOPS

while <condition>:

<expression>

<expression>

...

<condition> evaluates to a Boolean

if <condition> is True, do all the steps inside the while code block

check <condition> again

repeat until <condition> is False

CONTROL FLOW: while and for LOOPS

iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
print(n)
n = n+1
# shortcut with for loop
for n in range(5):
print(n)
```

CONTROL FLOW: for LOOPS

for <variable> in range(<some_num>):
<expression>
<expression>
...
each time through the loop, <variable> takes a value
first time, <variable> starts at the smallest value
next time, <variable> gets the prevvalue + 1
etc.

range(start,stop,step)

default values are start = 0and step = 1and optional
loop until value is stop -1

```
mysum= 0
for iin range(7, 10):
mysum+= i
print(mysum)
mysum= 0
for iin range(5, 11, 2):
mysum+= i
print(mysum)
```

break STATEMENT

immediately exits whatever loop it is in

skips remaining expressions in code block

exits only innermost loop!

```
while <condition_1>:
while <condition_2>:
<expression_a>
break
<expression_b>
<expression_c>
```

break STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```