LECTURE 2

DOHYUNG KIM

WHAT IS DISCUSSED IN THE LAST CLASS

- Preliminaries
  - Comment
  - Errors
  - Console output
  - Console Input
  - Import module

TODAY, WE WILL LEARN ABOUT

- Object and Expression
  - Built-in types, constants, functions
  - Built-in operators
    - Operator order
    - Floating point errors
    - Short-cut evaluation
    - Type-checking

OBJECTS

- Program works with data
- Every piece of data in a Python program is an **object**
- Every object has a **type**

BASIC BUILT-IN TYPE

- Examples

```
import math
def func():
  print("this is a user-defined function")
  return 4

print(type(1))
print(type(1.1))
print(type("1.1"))
print(type(1 < 2))
print(type(math))
print(type(math.sin))
print(type(func))
print(type(type(21)))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'module'>
<class 'builtin_function_or_method'>
<class 'function'>
<class 'type'>
>
```

SOME OTHER BUILT-IN TYPES

- We may see these types as well in this course

```
print(type(Exception()))
print(type(range(5)))
print(type([1,2,3]))
print(type((1,2,3)))
print(type({1,2}))
print(type({1:42}))
print(type(2+3j))
```

```
<class 'Exception'>
<class 'range'>
<class 'list'>
<class 'tuple'>
<class 'set'>
<class 'dict'>
<class 'complex'>
```

BUILT-IN CONSTANTS

- Constant represent a fixed value

```
import math

print(True)
print(False)
print(None)
print(math.pi)
print(math.e)
```

```
True
False
None
3.141592653589793
2.718281828459045
>
```

BUILT-IN FUNCTIONS

- Some functions can be used without definition or module importing

```
print(int(1.2))
print(float(21))
print(bool(0))

print(abs(-4))
print(max(4,3))
print(min(2,3))
print(pow(2,3))
print(round(1.274, 2))
```

```
1
21.0
False
4
4
2
8
1.27
>
```

BUILT-IN OPERATORS

- Arithmetic operators
  - +, -, *, /, //, **, %, - (unary), + (unary)
- Relational operators
  - <, <=, >=, >, ==, !=
- Assignment operators
  - +=, -=, *=, /=, //=, **=, %=, <<=, >>=
- Logical operators
  - and, or, not

ARITHMETIC OPERATORS

```
a = 15
b = 2

print("a + b =", a+b)
print("a - b =", a-b)
print("a / b =", a/b)
print("a // b =", a//b)
print("a * b =", a*b)
print("a ** b =", a**b)
print("a % b =", a%b)
print("+a =", +a)
print("-a =", -a)
```

```
a + b = 17
a - b = 13
a / b = 7.5
a // b = 7
a * b = 30
a ** b = 225
a % b = 1
+a = 15
-a = -15
```

- The operator / for normal float division
e.g., 15 / 2 results in 7.5

- The operator // for integer division
e.g., 15 // 2 results in 7

- The operator % for getting remainder of the division
e.g., 15 % 2 result in 1

ARITHMETIC OPERATORS

- More about the % operator

```
print(" 6 % 3 =", ( 6%3))
print(" 5 % 3 =", ( 5%3))
print(" 0 % 3 =", ( 0%3))
print("-4 % 3 =", (-4%3))
print(" 3 % 0 =", ( 3%0))
```

```
 6 % 3 = 0
 5 % 3 = 2
 0 % 3 = 0
-4 % 3 = 2
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    print(" 3 % 0 =", ( 3%0))
ZeroDivisionError: integer division or modulo by zero
>
```

- How can you make % using other arithmetic operators?
  - **a % b** is equivalent to **a - (a//b)*b**

RELATIONAL OPERATORS

```
val1 = 15
val2 = 2
val3 = 2

print(val1 <  val2)
print(val2 <= val3)
print(val1 >= val2)
print(val2 >  val3)
print(val1 == val2)
print(val2 != val3)
```

```
False
True
True
False
False
False
>
```

ASSIGNMENT OPERATORS

```
val1 = 15
val2 = 10

val1 +=  5
print("After val1 += 5, val1:", val1)
val2 -=  1
print("After val2 -= 1, val2:", val2)
val1 *=  2
print("After val1 *= 2, val1:", val1)
val1 /=  4
print("After val1 /= 4, val1:", val1)
val2 //= 2
print("After val2 //= 2, val2:", val2)
val1 **= 2
print("After val1 **= 2, val1:", val1)
val2 %=  3
print("After val2 /= 3, val2:", val2)
```

```
After val1 += 5  20
After val2 -= 1  9
After val1 *= 2  40
After val1 /= 4  10.0
After val2 //= 2  4
After val1 **= 2  100.0
After val2 /= 3  1
```

LOGICAL OPERATORS

```
val1 = True
val2 = False

print("val1 and val2 :", val1 and val2)
print("val1 or  val2 :", val1 or val2)
print("not val1 :", not val1)
```

SPECIAL TYPES OF OPERATORS

- Identity operators
  - is, is not
- Membership operators
  - in, not in
- These operators will be covered later

OPERATOR SEMANTICS

- Some operators can have different semantics depending on data type

```
print(1 * 2)
print(3 * "abc")
print(4 + 5)
print("abc" + "def")
print(3 + "def")
```

OPERATOR ORDER

- Precedence
  - % has same precedence as *, /, and //
  - ** has higher precedence than *, /, //, and %

```
print(1+2*3)
print(1+2%3)
print(2**3*4)
```

```
7
3
32
```

- Associativity
  - - associates left-to-right
  - ** associates right-to-left

```
print(1-2-3)
print(2**2**3)
```

```
-4
256
```

FLOATING POINT ERROR

- Something wired

```
> 1.2 - 1.0
0.19999999999999996
>
```

```
print(0.1 + 0.1 == 0.2)
print(0.1 + 0.1 + 0.1 == 0.3)
print(0.1 + 0.1 + 0.1)
print((0.1 + 0.1 + 0.1) - 0.3)
```

```
True
False
0.30000000000000004
5.551115123125783e-17
>
```

- What can you do if you want to get **0** for the equation 0.1 + 0.1 + 0.1 - 0.3

SHORT-CUT EVALUATION

```
def yes():
    return True

def no():
    return False

def crash():
    return 1/0 # error!

print(yes () or crash())
print(no() or crash())
print(crach() or no())
```

True
Error
Error

Why?

PRACTICE

```
def isPositive(n):
  result = (n > 0)
  print("isPositive(",n,") =", result)
  return result

def isEven(n):
  result = (n % 2 == 0)
  print("isEven(",n,") =", result)
  return result

print("Test 1: isEven(4) and isPositive(4))")
print(isEven(4) and isPositive(4))
print("----------")
print("Test 2: isEven(3) and isPositive(3)")
print(isEven(3) and isPositive(3))
```

TYPE CHECKING

- **type** and **isinstance** can be used to do type-checking
- **isinstance(x, T)** is more robust than **type(x) == T**
  - **isinstance** will be re-discussed when we cover top

```
val = "abc"
print(type(val) == str)
print(isinstance(val, str))

import numbers
val = 1+2j
print(type(val) == complex)
print(isinstance(val, numbers.Number))
print(isinstance(2.4, numbers.Number))
```

```
True
True
True
True
True
>
```

QUESTION?