Lecture 2 Variables, expressions, statements

IDS 400
Programming for Data Science in Business

Piazza

- Please login and check your permission as students **ASAP**.

Lab sessions

Lab 1 is available on Blackboard.

- You can start working on it at any time.
- You are **strongly encouraged to watch/practice all the lab sessions**.
- Without submitting 6 or more labs, you will automatically fail this course.

Lab sessions

- ***Struggling with the solution***

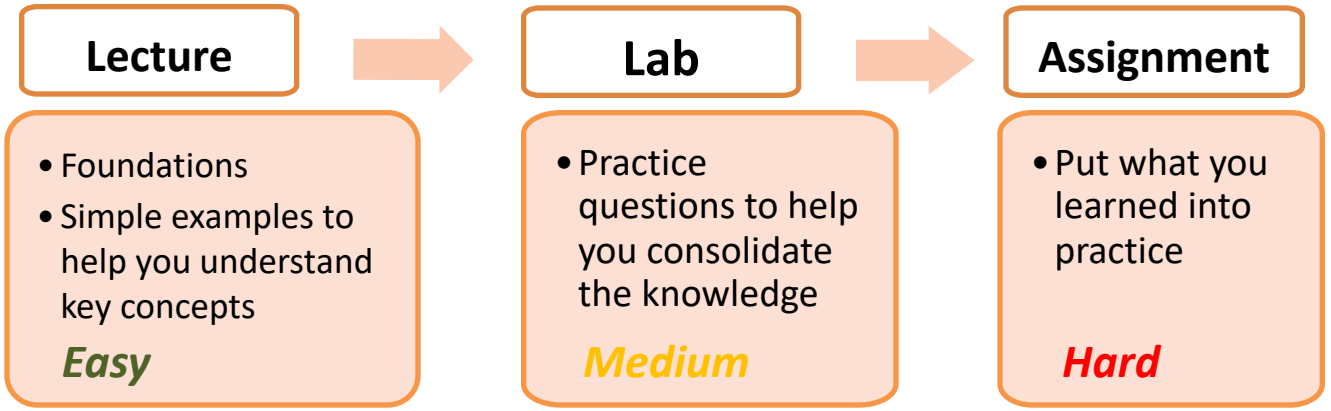
  You can skip the most difficult task. But please make sure to post your questions/confusions on Piazza, so that you can have a discussion with your classmates, and I know what we need to recap.

- ***If you come up with a good solution***

  Please DO post your good solutions on Piazza and share it with us!

(All above will earn you bonus)

Practice chain



Tentative schedule

Date	Lecture Number	Topics
08/24	Lecture 1	Introduction
08/31	Lecture 2	Basic
09/07	Lecture 3	Condition
09/14	Lecture 4	Loop
09/21	Lecture 5	String + **Quiz 1** → Online
09/28	Lecture 6	Type
10/05	Lecture 7	Function
10/12	Lecture 8	File + **Quiz 2** → Online
10/19	Lecture 9	Pandas
10/26	Lecture 10	Numpy
11/02	Lecture 11	Machine Learning
11/09	Lecture 12	Visualization
11/16	Lecture 13	Web Scraping & Deep Learning
11/23	***Thanksgiving***	***No lecture***
11/30	**Final presentation**	In class presentation
12/05	**Project submission due**	

Last class

- Python intro
- Install Python
- Print Statement in Python
- Comments in Python
- Program Flow in Python
    - Sequential
    - Conditional
    - Repeated

For This Class

- Variables
- Expression & Statements
- Operators

Variable name rules

- Must start with a letter or underscore (Not numbers)
- Only consist of letters, numbers, and underscores
- Case sensitive

smith Smith SMITH SmiTH

- You can not use reserved words as variable names

Question: Which of them are good names?

smith	$smiths	smith23	_smith	_23_	23smith	smith.23	a+b	smiTH	-smith

Variable name rules

- Must start with a letter or underscore (Not numbers)
- Only consist of letters, numbers, and underscores
- Case sensitive

smith Smith SMITH SmiTH

- You can not use reserved words as variable names

Question: Which of them are good names?

smith	$smiths	smith23	_smith	_23_	23smith	smith.23	a+b	smiTH	-smith
√	×	√	√	√	×	×	×	√	×

Variable name rules

- Must start with a letter or underscore
- Only consist of letters, numbers, and underscores
- Case sensitive

smith Smith SMITH SmiTH

- You can not use reserved words as variable names

Question: Which of them are good names?

smith	$smiths	smith23	_smith	_23_	23smith	smith.23	a+b	smiTH	-smith
√	×	√	√	√	×	×	×	√	×

```
32smith = 'hello world'   #must start with a letter or underscore
```

```
  File "<ipython-input-10-c91dd322161f>", line 1
    32smith = 'hello world'
          ^
SyntaxError: invalid syntax
```

Reserved words (python3)

- Reserved words (also called keywords) are defined with predefined meaning and syntax in the language.
- Reserved words **can not** be used as identifiers for other programming elements like name of variable, function etc.

```
class = 27  # can not use reserved words
```

```
  File "<ipython-input-11-7770a387218c>", line 1
    class = 27
          ^
SyntaxError: invalid syntax
```

Reserved words (Python 3)

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

- The above keywords may get altered in different versions of Python. Some extra might get added or some might be removed.

```
# You can always get the list of keywords in your current version
import keyword
print(keyword.kwlist)
```

Variable type

- The program can identify the type of a variable.
- You do not need to explicitly define or declare the type of a variable.

  int x =3 → most other languages

  x = 3 → Python

- Python has five standard data types
  - Number
  - String
  - List
  - Tuple
  - Dictionary

1- Numbers

- Python supports different numerical types
  - **int** -- signed integers, positive or negative whole numbers with no decimal point
  - **float** -- real numbers, positive or negative, written with a decimal point
  - **complex** -- consists of two parts, real and imaginary (denoted by j).

Here are some examples of numbers:

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0×260	-32.54e100	3e+26J
0×69	70.2-E12	4.53e-7j

2- Strings

- Strings in Python are identified as a contiguous set of characters represented in the single ‘ or double quotes “.
- Subsets of strings can be taken using the slice operator [ ] and [:] with **indexes starting at 0** in the beginning of the string.

```
str= 'HelloWorld!'

# Prints complete string
print(str)
```

```
HelloWorld!
```

```
# Prints first character of the string
print(str[0])
```

```
H
```

```
# Prints characters starting from 3rd to 5th
print(str[2:5])
```

```
llo
```

```
# Prints string starting from 3rd character
print(str[2:])
```

```
lloWorld!
```

3- Lists

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets [].
- The values stored in a list can be accessed using the slice operator [ ] and [:] with indexes starting at 0 in the beginning of the list.

```
list_example = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

# Prints complete list
print(list_example)
```

```
['abcd', 786, 2.23, 'john', 70.2]
```

```
# Prints firstelement of the list
print(list_example[0])
```

```
abcd
```

```
# Prints elements starting from 2nd till 3rd
print(list_example[1:3])
```

```
[786, 2.23]
```

```
# Prints elements starting from 3rd element
print(list_example[2:])
```

```
[2.23, 'john', 70.2]
```

Some Fun things about Slicing (:)

```
list_example = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

- print(list_example[-1])
- print(list_example[1:-1])
- list_example[0:2] = ‘z’

  print(list_example)

Some Fun things about Slicing (:)

```
list_example = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

- print(list_example[-1])

```
print(list_example[-1])
```

```
70.2
```

- print(list_example[1:-1])

```
print(list_example[1:-1])
```

```
[786, 2.23, 'john']
```

- list_example[0:2] = ‘z’

print(list_example)

```
list_example[0:2] = 'z'
print(list_example)
```

```
['z', 2.23, 'john', 70.2]
```

4- Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple contains items separated by commas enclosed within parentheses ().
- Difference between lists and tuples: Elements and size in lists can be changed, while tuples can not.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
```

```
print(tuple)# Prints complete list
```

```
('abcd', 786, 2.23, 'john', 70.2)
```

```
print(tuple[0])# Prints firstelement of the list
```

```
abcd
```

```
print(tuple[1:3])# Prints elements starting from 2nd till 3rd
```

```
(786, 2.23)
```

```
print(tuple[2:])# Prints elements starting from 3rd element
```

```
(2.23, 'john', 70.2)
```

Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple contains items separated by commas enclosed within square parentheses ().
- Difference between lists and tuples: Elements and size in lists can be changed, while tuples can not.

```
tuple[0:2] = 'z'
print(tuple)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-9a5a47ef5f22> in <module>
----> 1 tuple[0:2] = 'z'
      2 print(tuple)

TypeError: 'tuple' object does not support item assignment
```

5- Dictionary

- Consists of a number of key-value pairs.
- It is enclosed by curly braces { }

We will see more details about dictionary later.

Get variable type

- If you are not sure the type of a variable, the interpreter can tell you by using **type**.

```
type('Hello, world!')
```
```
str
```
```
type(17)
```
```
int
```
```
type(3.2)
```
```
float
```
```
type('4.7')
```
```
str
```

Type matters

- Python knows “type” of everythings.
- Some operations are prohibited. For example, you can not add a string to an integer.

```
a = '123'
b = a + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-3030f59d10f4> in <module>
      1 a = '123'
----> 2 b = a + 1

TypeError: can only concatenate str (not "int") to str
```

Type matters

- Operations on the same type operands will lead to results with the same type.

int + int => int

int – int => int

int * int => int

int / int => ?

Type matters

- Operations on the same type operands will lead to results with the same type.

int + int => int

int – int => int

int * int => int

int / int => ?

```
a = 123
b = a + 1
print(b)
```

```
124
```

```
print(a/b)
```

```
0.9919354838709677
```

```
print(b/2)
```

```
62.0
```

In Python 3, the result of division **/** is always float.

Type conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float.
- You can control this with the built-in functions **int()** and **float()**

```
i = 42
type(i)
```

```
int
```

```
f = float(i)
print(f)
type(f)
```

```
42.0
```

```
float
```

```
print(1 + 2 * 3 / 4 - 5)
```

```
-2.5
```

```
print(int(1 + 2 * 3 / 4 - 5))
```

```
-2
```

String conversions

- You can also use **int()** and **float()** to convert strings to integers/floating points.
- You will get an error if the string does not contain numeric characters.

```
sval = '123'
type(sval)
```

```
str
```

```
print(sval + 1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-844c8fe77d11> in <module>
----> 1 print(sval + 1)

TypeError: can only concatenate str (not "int") to str
```

```
ival = int(sval)
type(ival)
```

```
int
```

```
print(ival+ 1)
```

```
124
```

```
nsv = 'bob123'
niv = int(nsv)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-52-5ca314b9c43b> in <module>
      1 nsv = 'bob123'
----> 2 niv = int(nsv)

ValueError: invalid literal for int() with base 10: 'bob123'
```

User input

- We can ask Python to pause and read data from the keyboard using **input()** function.
- The **input()** function returns a **string**.

```
name = input("What's your name?")
```

```
What's your name?Aida
```

```
print('Welcome', name)
```

```
Welcome Aida
```

Converting user input

- If we want to read a number from the keyboard, we must convert it from a string to a number using a type conversion function **int()** or **float()**.

```
inp= input('Europe floor?')
```

```
Europe floor?4
```

```
usf = int(inp) + 1
```

```
print('US floor', usf)
```

```
US floor 5
```

The assignment statement

- An **assignment** statement creates new variables and gives them values.
- An assignment statement consists of *an expression on the right-hand side* and *a variable to store the result*.

```
>>> message = “hello, world”
>>> x = 17
>>> pi = 3.14159
```

Expression

- An expression is a combination of *values, variables and operators* to perform computation.
- Expressions need to be evaluated. If you ask Python to print an expression, the interpreter evaluates the expression and displays the result.

```
>>> 1 + 2 *7
15
>>> a = 10
>>> b = 20
>>> a + b
30
```

Statement

- A statement is an instruction that the Python interpreter can execute.
- We have seen two kinds of statements: **print** and **assignment**.
- When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

```
>>> a = 10
>>> b = 20
>>> print(a + b)
30
```

Expression VS Statement

- Expressions represent something. Expressions are combinations of values and operators and evaluate down to a single value.
- Anything that does something is a statement.

```
>>> 1 + 2 *7
15
>>> a = 10
>>> b = 20
>>> a + b
30
```

Operators

- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator uses are called operands.
- When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.
- Type of operator
  - Arithmetic operators
  - Comparison (Relational) operators
  - Assignment operators
  - Logical operators
  - Bitwise operators
  - Membership operators
  - Identity operators

Arithmetic operators

- Assume variable a has value 10 and b holds 20.

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2.0
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

The modulus operator

- It works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second.
- You can check whether one number is divisible by another – *if x % y is zero, then x is divisible by y*.
- You can extract the right-most digit or digits from a number.

```
isdivisible = 6 % 3
print(isdivisible)
```

```
0
```

```
isdivisible = 7 % 3
print(isdivisible)
```

```
1
```

```
lasttwodigits = 123491 % 100
print(lasttwodigits)
```

```
91
```

```
lastonedigits = 123491 % 10
print(lastonedigits)
```

```
1
```

Comparison operators

- Compare values on either sides of them and decide the relation among them.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Discarded in Python 3

```
x = 10
y = 20

# Output: x > y is False
print('x > y is',x>y)

# Output: x < y is True
print('x < y is',x<y)

# Output: x == y is False
print('x == y is',x==y)

# Output: x != y is True
print('x != y is',x!=y)

# Output: x >= y is False
print('x >= y is',x>=y)

# Output: x <= y is True
print('x <= y is',x<=y)
```

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

Assignment operators

- Assignment operators are used to assign values to variables.
- There are various compound operators in Python like **a += 5** that adds 5 to the variable a. It is equivalent to **a = a + 5**.

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Bitwise operators

- Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit; hence the name is bitwise.

a = 60
b = 13

Binary representation:
a = 00111100
b = 00001101

a&b = 00001100

a|b = 00111101

a^b = 00110001

~a = 11000011

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a<<2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Logical operators

- Logical operators are used to combine conditional statements.

Assume variable a has value ***True*** and b holds ***False***.

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True

```
x = True
y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)
```

```
x and y is False
x or y is True
not x is False
```

Membership operators (in/ not in)

- Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

```
x = 'Hello world'

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)
```

```
True
True
```

Identity operators

- **is** and **is not** are the identity operators in Python. They are used to check if two values (or variables) are *located on the same part of the memory*.
- Two variables that are equal does not imply that they are identical.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

```
x1 = 5
y1 = 5

x2 = 'Hello'
y2 = 'Hello'

x3 = [1,2,3]
y3 = [1,2,3]

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False
print(x3 is y3)
```

```
False
True
False
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

Operator precedence

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.
  - *Parentheses have the highest precedence* and can be used to force an expression to evaluate in the order you want.
  - *Exponentiation has the next highest precedence*.
  - Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence .
  - *Operators with the same precedence are evaluated from left to right*.

Operator precedence

- The operator precedence in Python is listed in the following table. It is in descending order (upper group has higher precedence than the lower ones).

Operators	Meaning
`()`	Parentheses
`**`	Exponent
`+x`, `-x`, `~x`	Unary plus, Unary minus, Bitwise NOT
`*`, `/`, `//`, `%`	Multiplication, Division, Floor division, Modulus
`+`, `-`	Addition, Subtraction
`<<`, `>>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

A simple example

```
>>> x = 6

>>> x = 4 * x ** (8 - x)

>>> print(x)
```

What is the output?

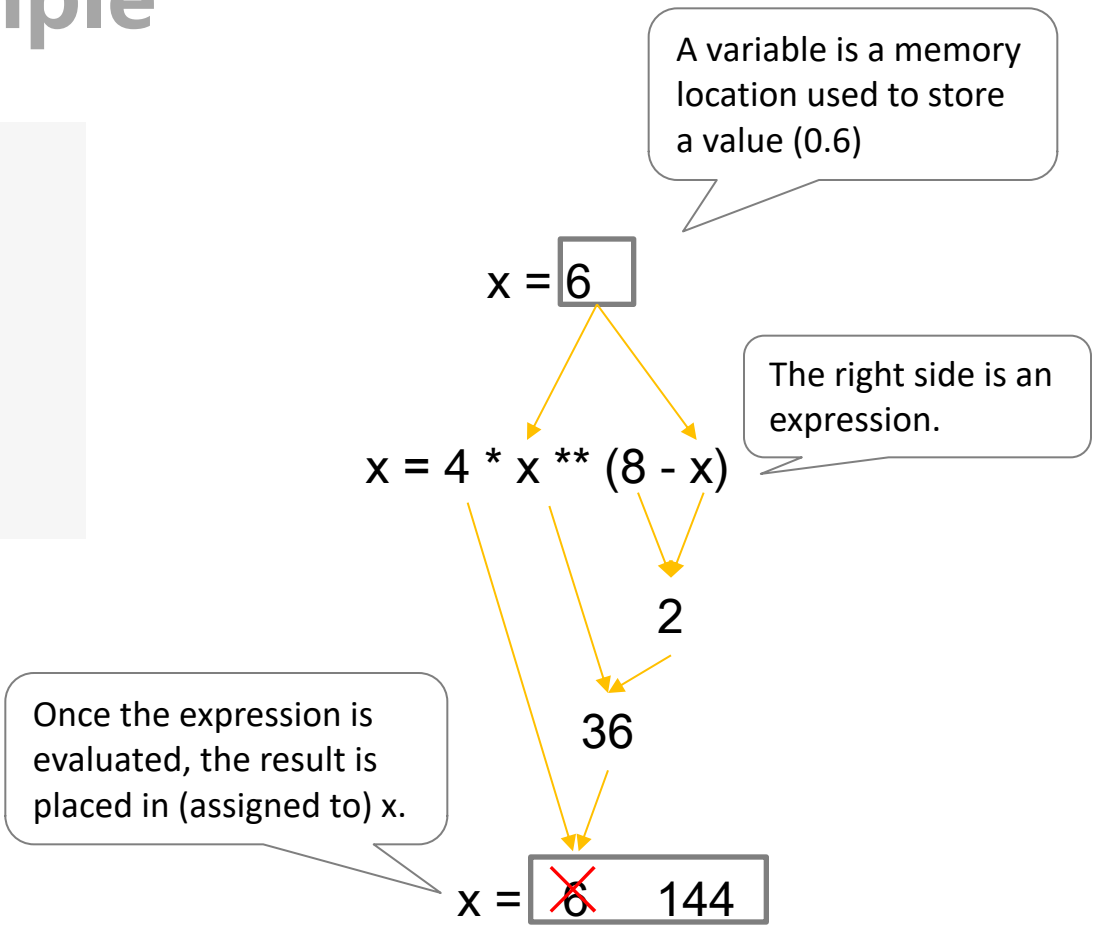
A simple example

```
>>> x = 6

>>> x = 4 * x ** (8 - x)

>>> print(x)
```

What is the output?



Summary

- Variables
  - Name rules
  - Types
- Expression & Statements
- Operators (types, precedence)
  - Arithmetic operators
  - Comparison (Relational) operators
  - Assignment operators
  - Logical operators
  - Bitwise operators
  - Membership operators
  - Identity operators

Lab *Basic*