

Variables and assignments

Variables

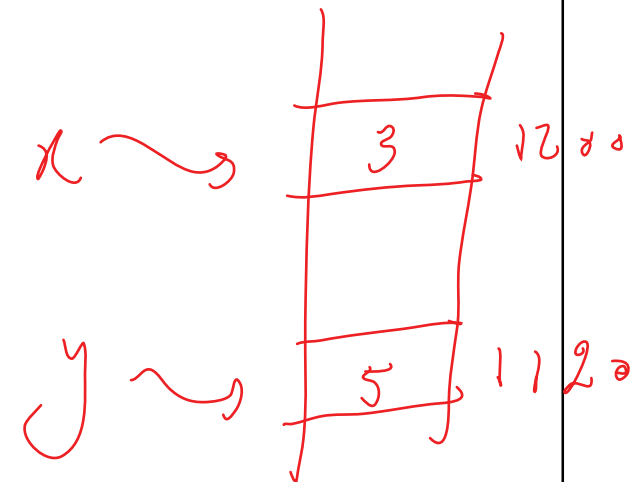
The billions of bytes of main storage in your home computer are used to store both machine instructions and data. The electronic circuits of main memory (and all other types of memory) make no distinction between the two. When a program is running, some memory locations are used for machine instructions and others for data. Later, when another program is running some of the bytes that previously held machine instructions may now hold data, and some that previously held data may now hold machine instructions. Using the same memory for both instructions and data was the idea of **John von Neumann**, a computer pioneer. (NOTE: if you are unclear what bytes and memory locations are, please read Chapter 3.)

Variables

- We can not access to memory directly
- Instead, we define some names as memory units
- Interpreter bind these names to memory address
- These names called **variables**
- **Variable is** a memory location which used for holding data and is defined as a name in the program.
- Examples:

X=3

Y=5



Variables

- Two kinds of variables
 - Ordinary variables
 - Reference variables
- Ordinary variables: a name for a location in memory which uses a particular data type to hold a value
- Reference variables: a name for a location in memory which point to another location of memory which uses a particular data type to hold a value

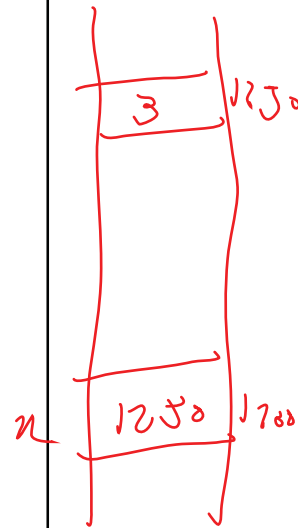
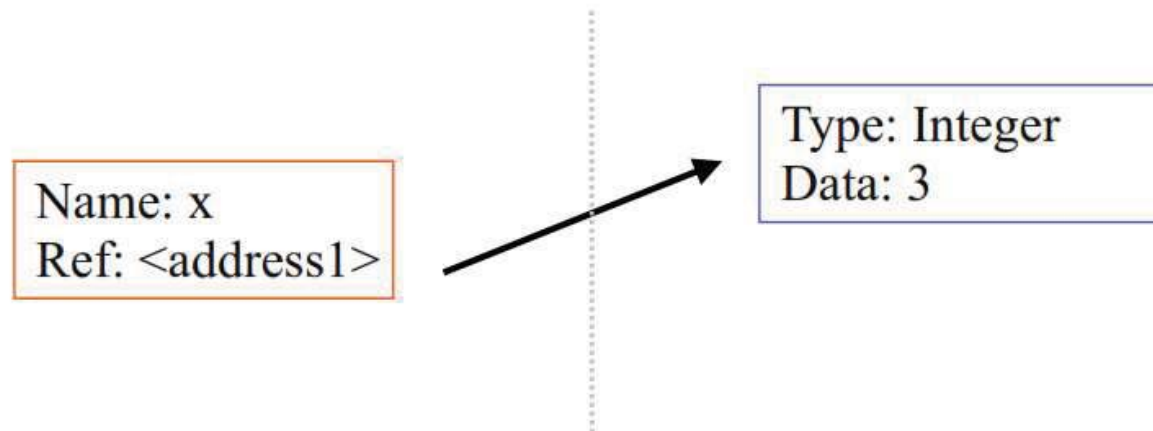
Variables

- Ordinary variables: a name for a location in memory which uses a particular data type to hold a value



Variables

- Reference variables: a name for a location in memory which point to another location of memory which uses a particular data type to hold a value



Assignment

- *Variables are created by assignment (=)*
- You create a name the first time it appears on the left side of an assignment expression:

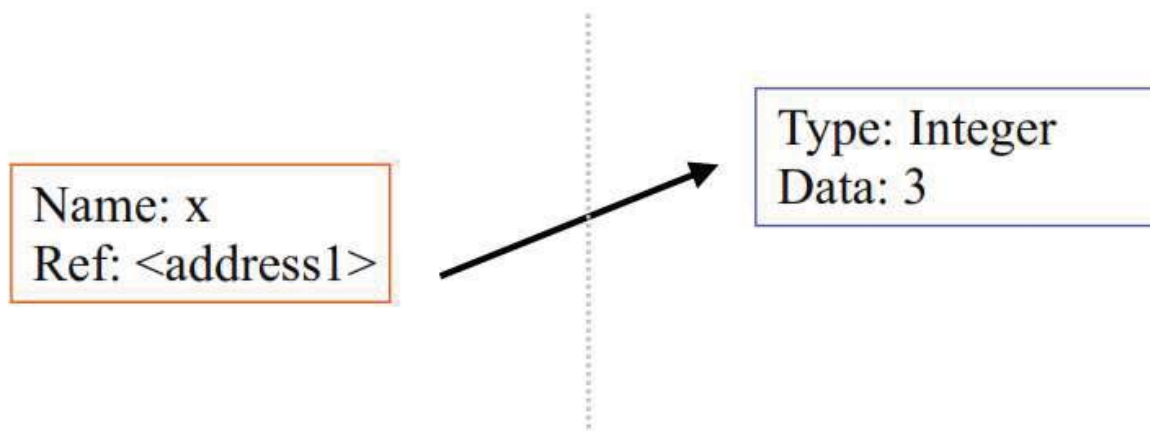
x = 3

Name: x
Ref: <address1>



Type: Integer
Data: 3

Assignment



- A reference is deleted via garbage collection after any names bound to it have passed out of scope

Assignment

- *Variables are created by assignment (=)*
- You create a name the first time it appears on the left side of an assignment expression:
`x = 3`
- Names in Python do not have an intrinsic type, objects have types
 - Python determines the type of the reference automatically based on what data is assigned to it
- A reference is deleted via garbage collection after any names bound to it have passed out of scope
- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*
 - *Assignment creates references, not copies*
- Python uses *reference semantics* (more later)

Type of variables

- **Type of a variable is determined by its value**
- **Python has many built-in data types. You have seen: int (integer), float (floating-point number), str (string), list (list), and dict (dictionary). They all have distinct representations.**

`X=3`

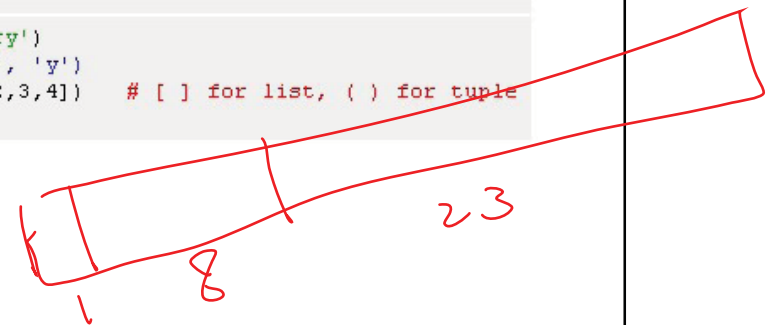
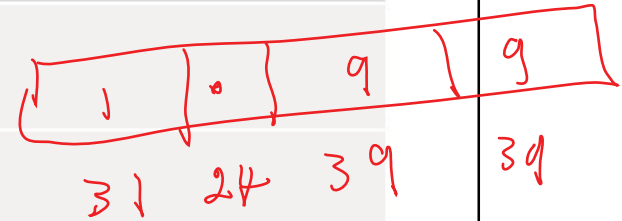
`Y=4.2`

`Str="this"`

- **A variable can be assigned by other variables or literals**

Type of variables

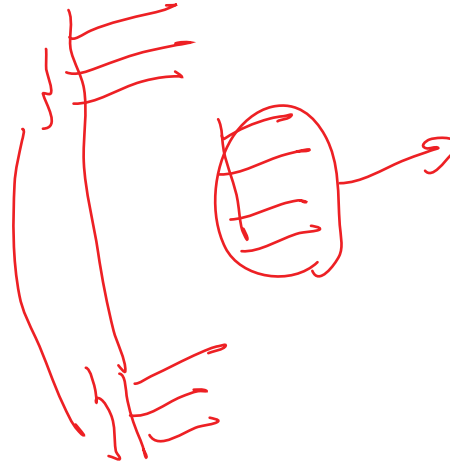
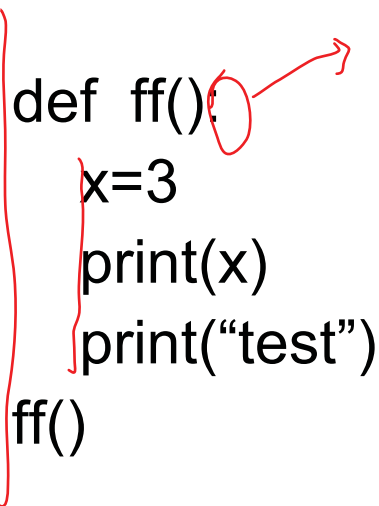
Function	Converting what to what	Example
<code>int()</code>	string, floating point → integer	<pre>>>> int('2014') 2014 >>> int(3.141592) 3</pre>
<code>float()</code>	string, integer → floating point number	<pre>>>> float('1.99') 1.99 >>> float(5) 5.0</pre>
<code>str()</code>	integer, float, list, tuple, dictionary → string	<pre>>>> str(3.141592) '3.141592' >>> str([1,2,3,4]) '[1, 2, 3, 4]'</pre>
<code>list()</code>	string, tuple, dictionary → list	<pre>>>> list('Mary') # list of characters in 'Mary' ['M', 'a', 'r', 'y'] >>> list((1,2,3,4)) # (1,2,3,4) is a tuple [1, 2, 3, 4]</pre>
<code>tuple()</code>	string, list → tuple	<pre>>>> tuple('Mary') ('M', 'a', 'r', 'y') >>> tuple([1,2,3,4]) # [] for list, () for tuple (1, 2, 3, 4)</pre>



Enough to Understand the Code

- Indentation matters to code meaning
 - Block structure indicated by indentation

```
def ff():  
    x=3  
    print(x)  
    print("test")  
ff()
```



- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Enough to Understand the Code

- First assignment to a variable creates it
 - Variable types don't need to be declared.
 - Python figures out the variable types on its own.

```
def ff():  
    x=3  
    print(x)  
    print("test")  
ff()
```

Enough to Understand the Code

- Assignment is `=` and comparison is `==`

```
def ff():  
    x=3  
    print(x)  
    print("test")  
ff()
```

Enough to Understand the Code

- For numbers $+$ $-$ $*$ $/$ $\%$ are as expected
 - Special use of $+$ for string concatenation and $\%$ for string formatting (as in C's printf)

```
def ff():
```

```
    x=3
```

```
    y=7
```

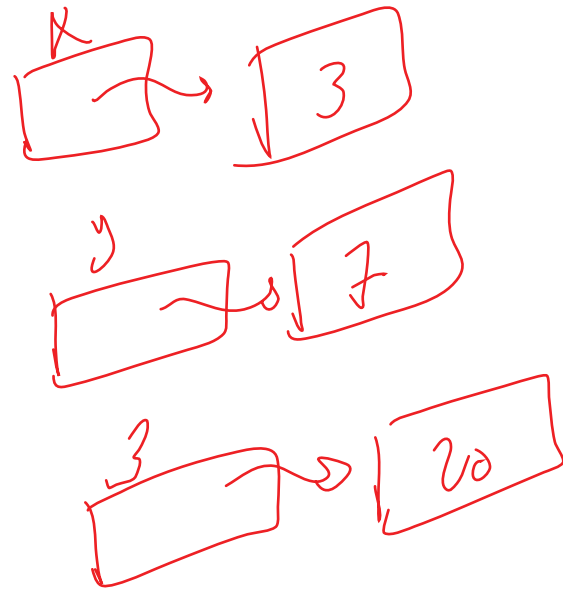
```
    z=x+y*2
```

```
    print(x, z)
```

```
    print("test" + "prog")
```

```
ff()
```

- $x+y*2$ is a expression



Enough to Understand the Code

- The basic printing command is `print`

```
def ff():  
    x=3  
    y=7  
    z=x+y*2  
    print(x, z)  
    print("test" + "prog")  
ff()
```


Comments

- Start comments with #, rest of line is ignored
- Can include a “documentation string” as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it’s good style to include one

```
# a simple program  
x=3  
Print(x)
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words:

and, assert, break, class, continue,
def, del, elif, else, except, exec,
finally, for, from, global, if,
import, in, is, lambda, not, or,
pass, print, raise, return, try,
while

2 A 5 3
A 2 5 3

Assignment

- You can assign to multiple names at the same time

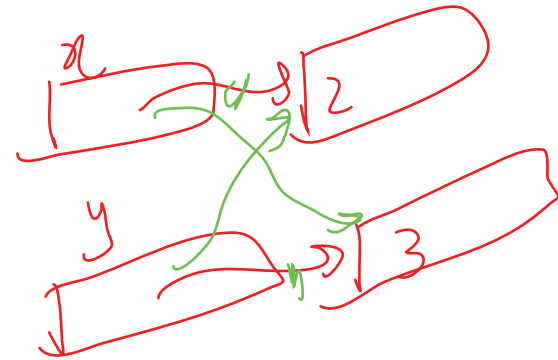
```
>>> x, y = 2, 3
```

```
>>> print(x)
```

```
2
```

```
>>> print(y)
```

```
3
```



This makes it easy to swap values

```
>>> x, y = y, x
```

- Assignments can be chained

```
>>> a = b = x = 2
```

Accessing Non-Existent Name

Accessing a name before it's been properly created (by placing it on the left side of an assignment), raises an error

```
>>> print(y)
```

```
Traceback (most recent call last):  
  File "<pyshell#16>", line 1, in -toplevel-  
    y
```

```
NameError: name 'y' is not defined
```

```
>>> y = 3
```

```
>>> print(y)
```

```
3
```

print

- **print** : Produces text output on the console.

- **Syntax:**

```
print ("Message")
```

```
print (Expression)
```

- Prints the given text message or expression value on the console, and moves the cursor down to the next line.

```
print (Item1, Item2, ..., ItemN)
```

- Prints several messages and/or expressions on the same line.

- **Examples:**

```
print ("Hello, world!")
```

```
age = 45
```

```
print ("You have", age, "years until retirement")
```

Output:

```
Hello, world!
```

```
You have 45 years until retirement
```

print

- **print** : Produces text output on the console.

- **Syntax:**

```
print ("Message", end=" ")  
print (Expression, end=" ")
```

- **Examples:**

```
print ("This is a test")  
print ("Hello, world!", end=" ")  
age = 45  
print ("You have", age)
```

Output:

```
This is a test  
Hello, world!   You have 45
```

type

- **type** : Produces the type of the variable

- **Syntax:**

`type(x)`

- **Examples:**

```
x=5
print (type(x))
x= "this"
print (type(x))
y= 3.14
print (type(y))
```

Output:

```
<class 'int'>
<class 'str'>
<class 'float'>
```

Operators and operands

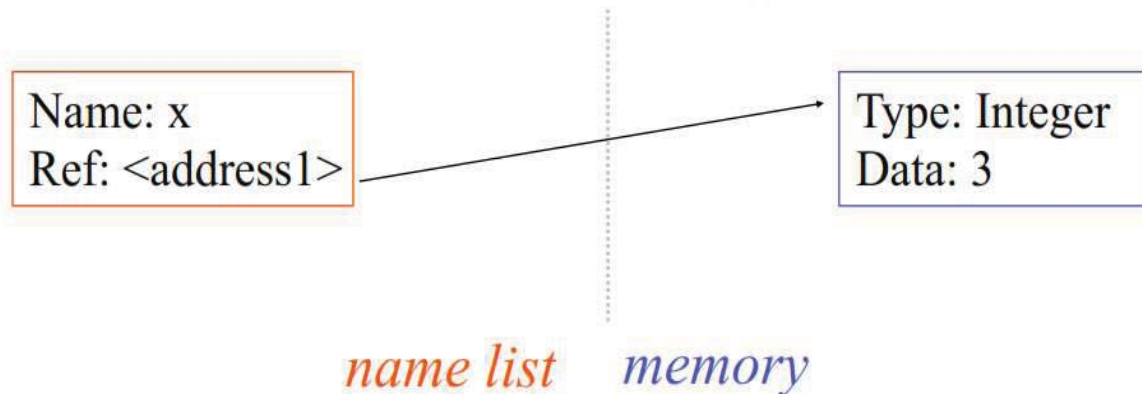
- **Operators are the constructs which can manipulate the data**
- **Data are named operand**
- **Expression is a combination of literals, operators, operands, variables and parenthesis used to calculate a value**
- **Consider the expression $4 + 5$. Here, 4 and 5 are called operands and + is called operator.**

Understanding Reference Semantics II

- There is a lot going on when we type:

$x = 3$

- First, an integer **3** is created and stored in memory
- A name **x** is created
- An *reference* to the memory location storing the **3** is then assigned to the name **x**
- So: When we say that the value of **x** is **3**
- we mean that **x** now refers to the integer **3**



Understanding Reference Semantics III

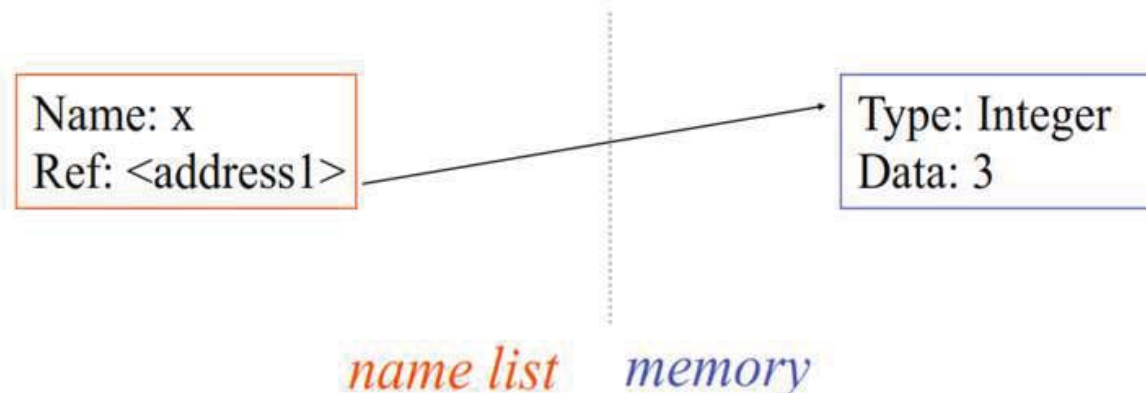
- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are “immutable.”
- This doesn't mean we can't change the value of *x*, i.e. *change what x refers to ...*
- For example, we could increment *x*:

```
>>> x = 3
>>> x = x + 1
>>> print(x)
4
```

Understanding Reference Semantics II

$x = 3$

- First, an integer **3** is created and stored in memory
- A name ***x*** is created
- An **reference** to the memory location storing the **3** is then assigned to the name ***x***

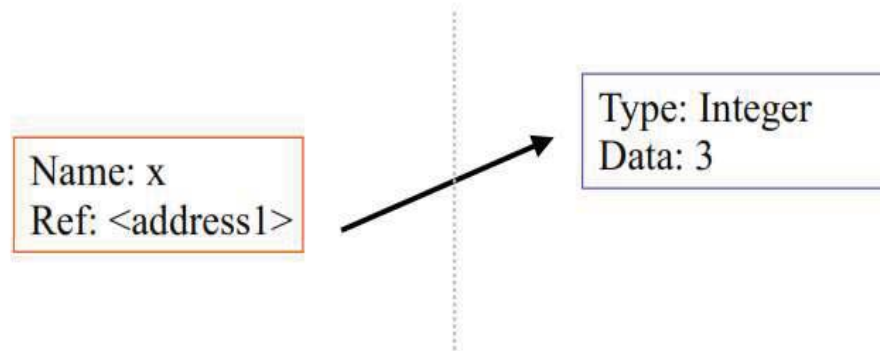



Understanding Reference Semantics IV

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.

```
>>> x = x + 1
```

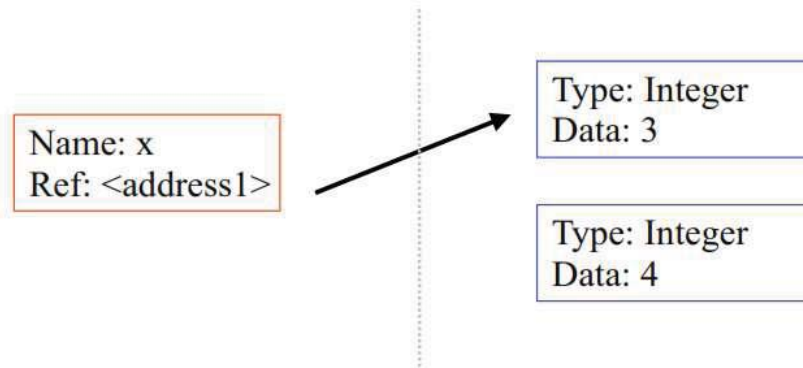


Understanding Reference Semantics IV

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.
3. *The $3+1$ calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.*

`>>> x = x + 1`

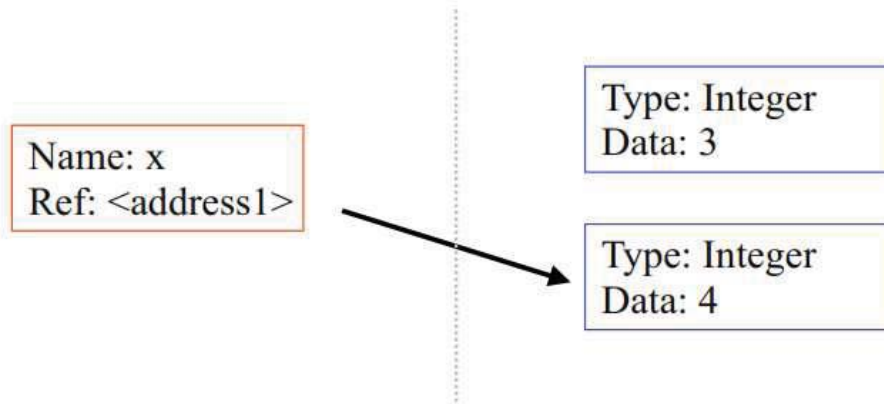


Understanding Reference Semantics IV

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.
3. The $3+1$ calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
4. *The name x is changed to point to this new reference.*

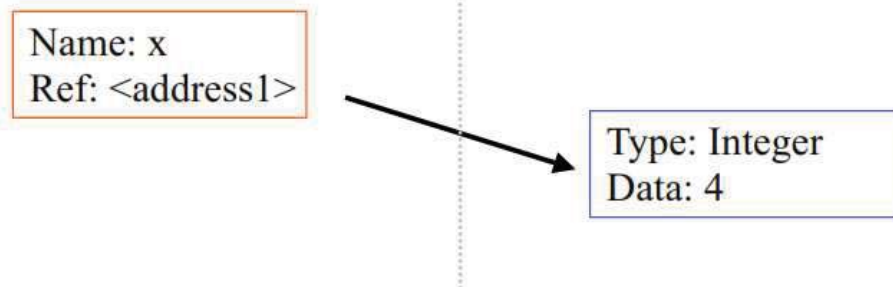
```
>>> x = x + 1
```



Understanding Reference Semantics IV

- If we increment **x**, then what's really happening is:

1. The reference of name **x** is looked up. >>> **x** = **x** + 1
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. The name **x** is changed to point to this new reference.
5. *The old data **3** is garbage collected if no name still refers to it.*



Assignment 1

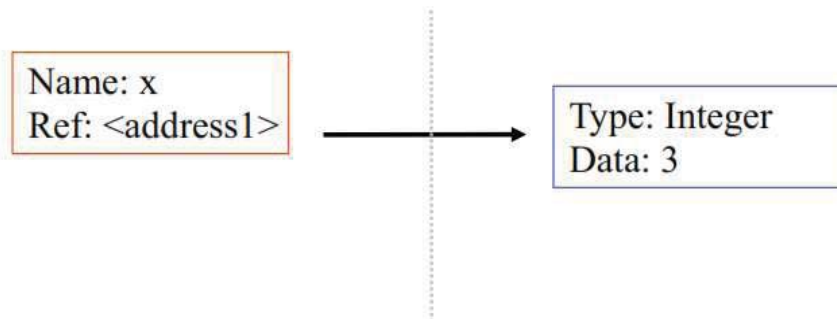
- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```


Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

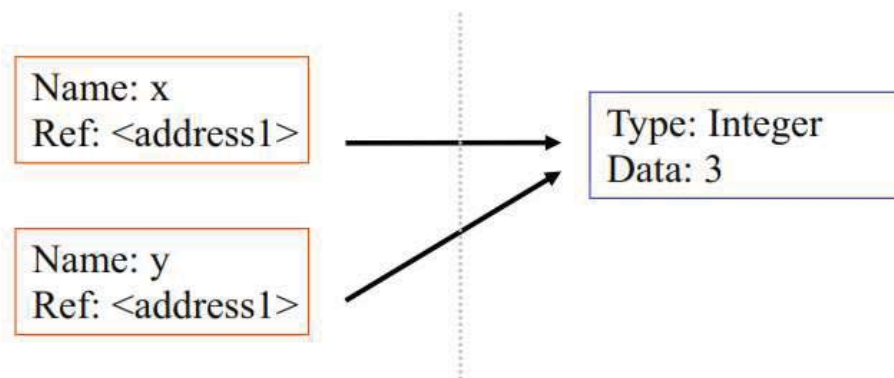
```
→ >>> x = 3          # Creates 3, name x refers to 3
    >>> y = x          # Creates name y, refers to 3.
    >>> y = 4          # Creates ref for 4. Changes y.
    >>> print(x)      # No effect on x, still ref 3.
    3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

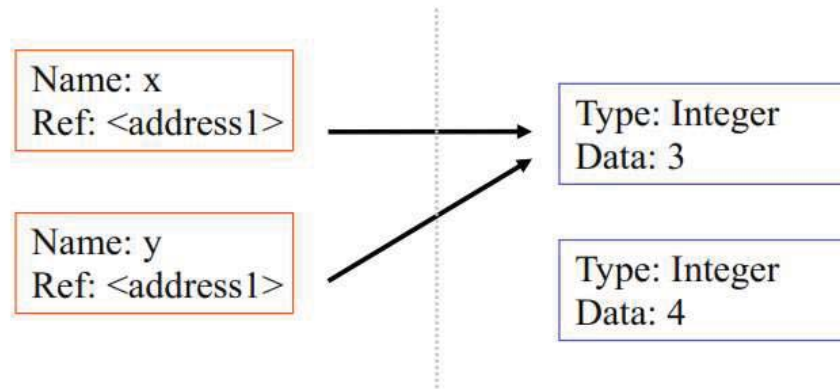
```
→ >>> x = 3          # Creates 3, name x refers to 3
    >>> y = x         # Creates name y, refers to 3.
    >>> y = 4         # Creates ref for 4. Changes y.
    >>> print(x)      # No effect on x, still ref 3.
    3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

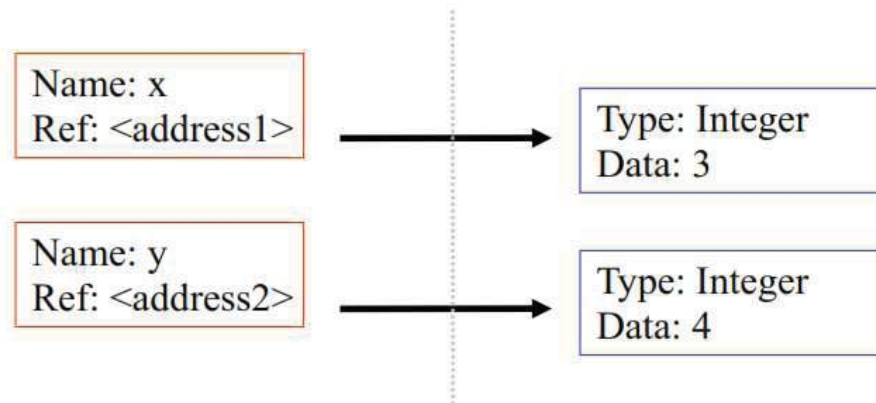
```
>>> x = 3          # Creates 3, name x refers to 3
→ >>> y = x         # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

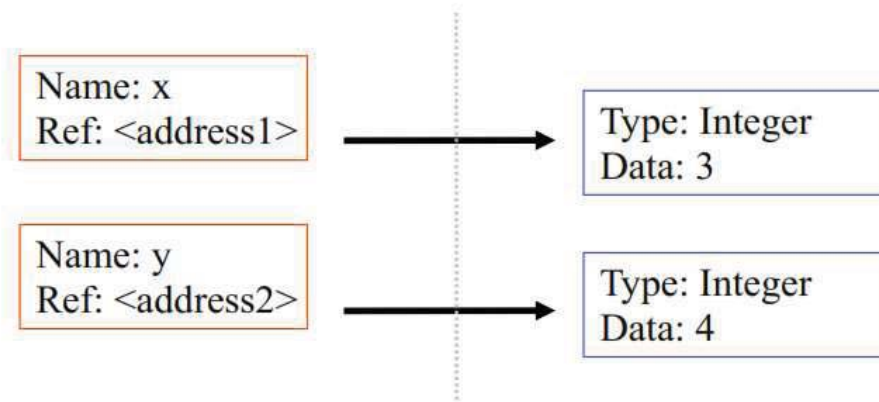
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
→ >>> y = 4        # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
→ >>> print(x)     # No effect on x, still ref 3.
3
```



Assignment 2

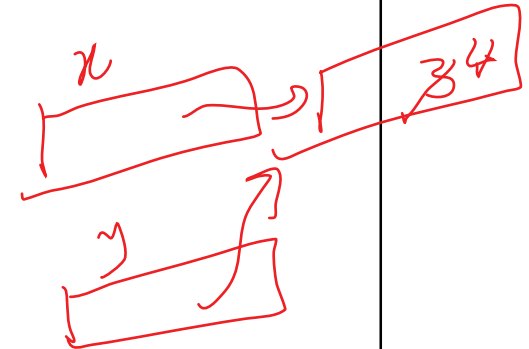
- For other data types (lists, dictionaries, user-defined types), assignment works differently.
 - These datatypes are “mutable.”
 - When we change these data, we do it *in place*.
 - We don't copy them into a new memory address each time.
 - If we type `y=x` and then modify `y`, both `x` and `y` are changed.

immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print(x)
3
```

mutable

```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```



Expression

- **Expression is a combination of literals, operators, operands, variables and parenthesis used to calculate a value**
- `X+y-r+myVar-3 + (5*w)`

Types of Operator

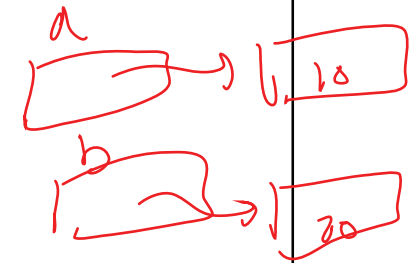
- **Python language supports the following types of operators.**
 - Arithmetic Operators
 - Comparison (Relational) Operators
 - Assignment Operators
 - Logical Operators
 - Bitwise Operators
 - Membership Operators
 - Identity Operators

Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then

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	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$, $-11 // 3 = -4$, $-11.0 // 3 = -4.0$

$a = 10$
 $b = 20$



$c = a + b$



$9 / 2 = 4.5$

Arithmetic Operators

```
a = 21
b = 10
c = 0
```

```
c = a + b
print ("Line 1 - Value of c is ", c)
```

```
c = a - b
print ("Line 2 - Value of c is ", c)
```

```
c = a * b
print ("Line 3 - Value of c is ", c)
```

```
c = a / b
print ("Line 4 - Value of c is ", c)
```

```
c = a % b
print ("Line 5 - Value of c is ", c)
```

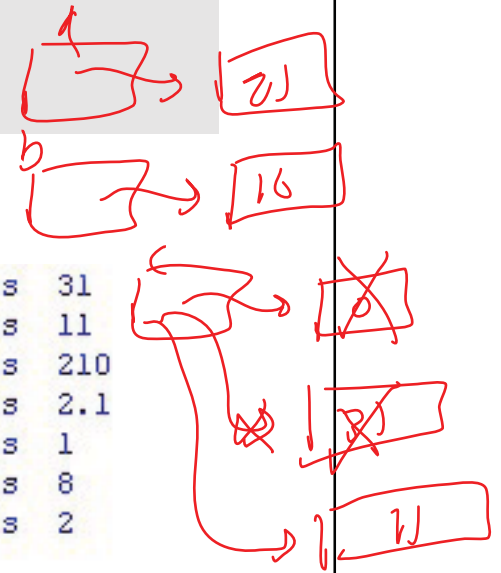
```
a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c)
```

```
a = 10
b = 5
c = a//b
print ("Line 7 - Value of c is ", c)
```

a/b

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2.1
Line 5 - Value of c is 1
Line 6 - Value of c is 8
Line 7 - Value of c is 2
>>>
```

a	b	c
21	10	0
2	3	21 2
10	5	11 2
		21 0
		2.1
		1 8
		2



Comparison Operators

- **These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.**

Assume variable a holds 10 and variable b holds 20, then –

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.	
<>	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.

if

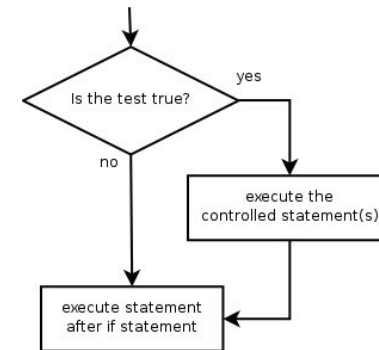
- **if statement:** Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.

- Syntax:

```
if condition:  
    statements
```

- **Example:**

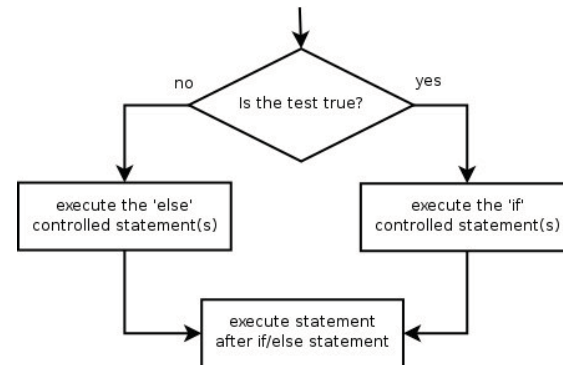
```
gpa = 3.4  
if gpa > 2.0:  
    print ("Your application is accepted.")
```



if/else

- **if/else statement:** Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

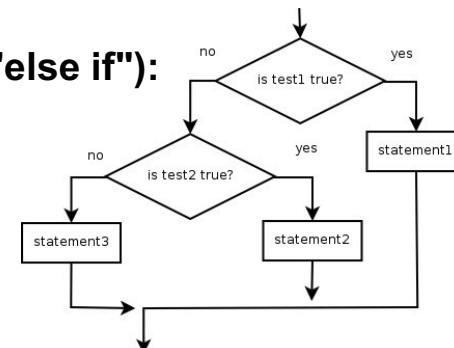
- Syntax:
if **condition**:
 statements
else:
 statements



- **Example:**
gpa = 1.4
if gpa > 2.0:
 print ("Welcome to Mars University!")
else:
 print ("Your application is denied.")

- **Multiple conditions can be chained with elif ("else if"):**

```
if condition:  
    statements  
elif condition:  
    statements  
else:  
    statements
```



Comparison Operators

hello1.py - C:/Documents and Settings/admin/Desktop/intro-python

File Edit Format Run Options Window Help

```
a = 21
b = 10
c = 0

if ( a == b ):
    print ("Line 1 - a is equal to b")
else:
    print ("Line 1 - a is not equal to b")

if ( a != b ):
    print ("Line 2 - a is not equal to b")
else:
    print ("Line 2 - a is equal to b")

if ( a <=> b ):
    print ("Line 3 - a is not equal to b")
else:
    print ("Line 3 - a is equal to b")

if ( a < b ):
    print ("Line 4 - a is less than b" )
else:
    print ("Line 4 - a is not less than b")

if ( a > b ):
    print ("Line 5 - a is greater than b")
else:
    print ("Line 5 - a is not greater than b")
```

a b c
21 10 0

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not equal to b
Line 4 - a is not less than b
Line 5 - a is greater than b
>>> |
```

Assignment Operators

Assume variable a holds 10 and variable b holds 20, then

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 $c = c / a$ $c /= a$ 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 $c = c // a$

a b c

 10 20 30
 40

$c += a$

$c = c + a$

Bitwise Operators

- Bitwise operator works on bits and performs bit by bit operation. Assume if $a = 60$; and $b = 13$; Now in binary format they will be as follows
- $a = 0011\ 1100$
- $b = 0000\ 1101$
- -----
- $a \& b = 0000\ 1100$
- $a | b = 0011\ 1101$
- $a \wedge b = 0011\ 0001$
- $\sim a = 1100\ 0011$

Bitwise Operators

- **a = 0011 1100**
- **b = 0000 1101**

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 >> 2; = 15 (means 0000 1111)

Handwritten binary calculations for a & b and a | b:

```

      0011 1100
    & 0000 1101
    -----
      0000 1100  (60)

      0011 1100
    | 0000 1101
    -----
      0011 1101  (61)
  
```

Bitwise Operators

hello1.py - C:/Documents and Settings/admin/Desktop/intro-python/e

File Edit Format Run Options Window Help

```
a = 60          # 60 = 0011 1100
b = 13          # 13 = 0000 1101
c = 0

c = a & b;      # 12 = 0000 1100
print ("Line 1 - Value of c is ", c)

c = a | b;      # 61 = 0011 1101
print ("Line 2 - Value of c is ", c)

c = a ^ b;      # 49 = 0011 0001
print ("Line 3 - Value of c is ", c)

c = ~a;         # -61 = 1100 0011
print ("Line 4 - Value of c is ", c)

c = a << 2;      # 240 = 1111 0000
print ("Line 5 - Value of c is ", c)

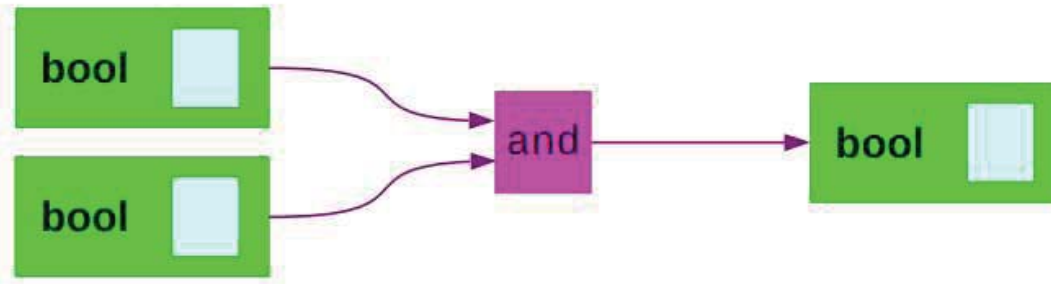
c = a >> 2;      # 15 = 0000 1111
print ("Line 6 - Value of c is ", c)
```

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Logical Operators

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
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 false.

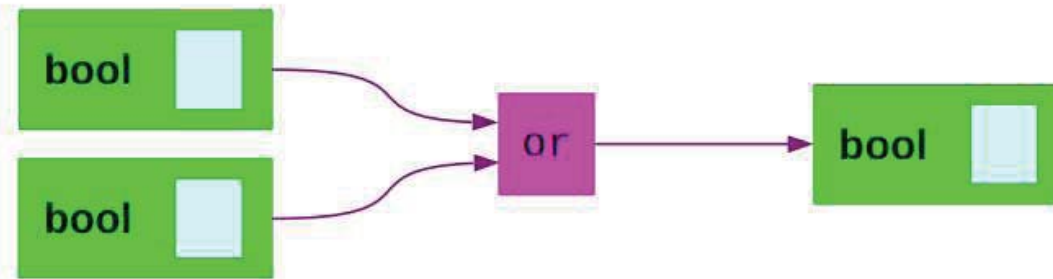
Boolean operations — “and”



True	and	True	→	True
True	and	False	→	False
False	and	True	→	False
False	and	False	→	False

Both have
to be True

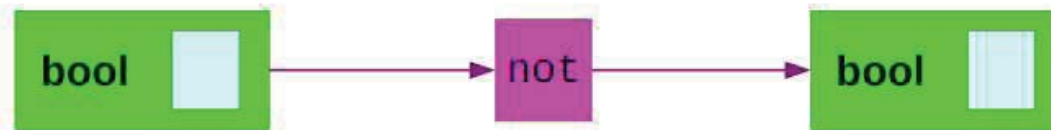
Boolean operations — “or”



True	or	True	→	True
True	or	False	→	True
False	or	True	→	True
False	or	False	→	False

At least
one has
to be True

Boolean operations — “not”



`not True` → `False`

`not False` → `True`

Logical Operators

```
a = 60  
b = 13  
c = 17
```

```
if (a>b) or (c>b) :  
    print ("the condition is true")  
else:  
    print ("the condition is true")
```

and
or
not

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

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 find 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.

s1 is 'this'
s2 is 'hi'

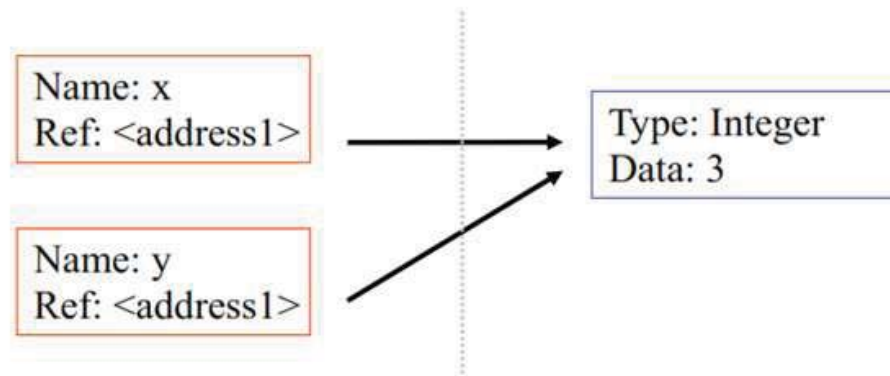
if (s2 in s1):

Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below

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).

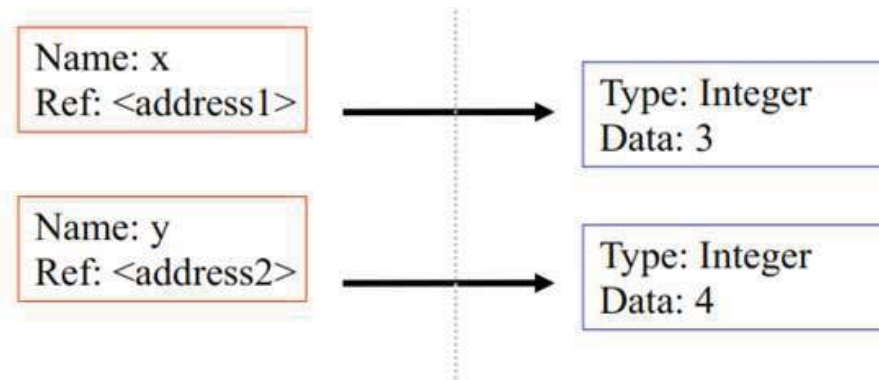
if (x is y):



Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below

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).



if (x is y):
if (x is not y):

Operators Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operators Precedence

hello1.py - C:/Documents and Settings/admin/Desktop/intro-python/e

File Edit Format Run Options Window Help

```
a = 20
b = 10
c = 15
d = 5
e = 0
```

```
e = a + b * c / d
print ("Value of a + b * c / d is ", e)
```

```
e = ((a + b) * c) / d      # (30 * 15) / 5
print ("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);    # (30) * (15/5)
print ("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;      # 20 + (150/5)
print ("Value of a + (b * c) / d is ", e)
```

```
Value of a + b * c / d is 50.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
>>>
```