Python Applications for Digital Design and Signal Processing

# Applications for Digital Design and Signal Processing
# Session 2

Dan Boschen

# Course Outline

| Session | Topics |
|---|---|
| **1** | Course Intro: Python, Spyder and Jupyter |
| 2 | **Core Python** |
| 3 | Core Python |
| 4 | Core Python |
| 5 | Python Modules and Packages |
| 6 | NumPy |
| 7 | NumPy, SciPy |
| 8 | Python for Verification, Modelling and Analysis |

# Session 2 Contents

**Goals for this Session:** Review of the core Python language

# Python Core

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

```
>>> import keyword
>>> print(keyword.kwlist)
```

# Python Built-In Functions

| Iterables / Iterators | Type Conversion | Math | Obj Creation | Variables / References | Class Constructs | Other |
|---|---|---|---|---|---|---|
| all() | ascii() | abs() | bytearray() | dir() | classmethod() | breakpoint() |
| any() | bin() | divmod() | bytes() | globals() | delattr() | callable() |
| enumerate() | bool() | max() | dict() | id() | getattr() | compile() |
| filter() | chr() | min() | frozenset() | locals() | hasattr() | eval() |
| iter() | complex() | pow() | list() | vars() | isinstance() | exec() |
| len() | float() | round() | object() | hash() | issubclass() | help() |
| map() | hex() | sum() | set() | | property() | memoryview() |
| next() | int() | | tuple() | | setattr() | staticmethod() |
| range() | oct() | **I/O** | | | super() | |
| reversed() | ord() | format() | | | | |
| slice() | repr() | input() | | | | |
| sorted() | str() | open() | | | | |
| zip() | type() | print() | | | | |

# Built-in Modules

| | |
|---|---|
| cmath: | mathematical functions for complex numbers |
| collections: | additional container data types |
| inspect: | extract info and source code from live objects |
| itertools: | functions creating iterators for efficient looping |
| math: | scientific calculations and constants |
| os: | operating system interfaces |
| random: | random number generators |
| sys: | Python runtime environment manipulation |

To see full list: https://docs.python.org/3/py-modindex.html

# Help!

```
>>> help()

>>> help("modules doc text to search")

>>> help(object)

>>> help("module, function, method")
```

https://docs.python.org/3/

# Attribute Access

Everything in Python is an **object**

Access object attributes using **dot notation**

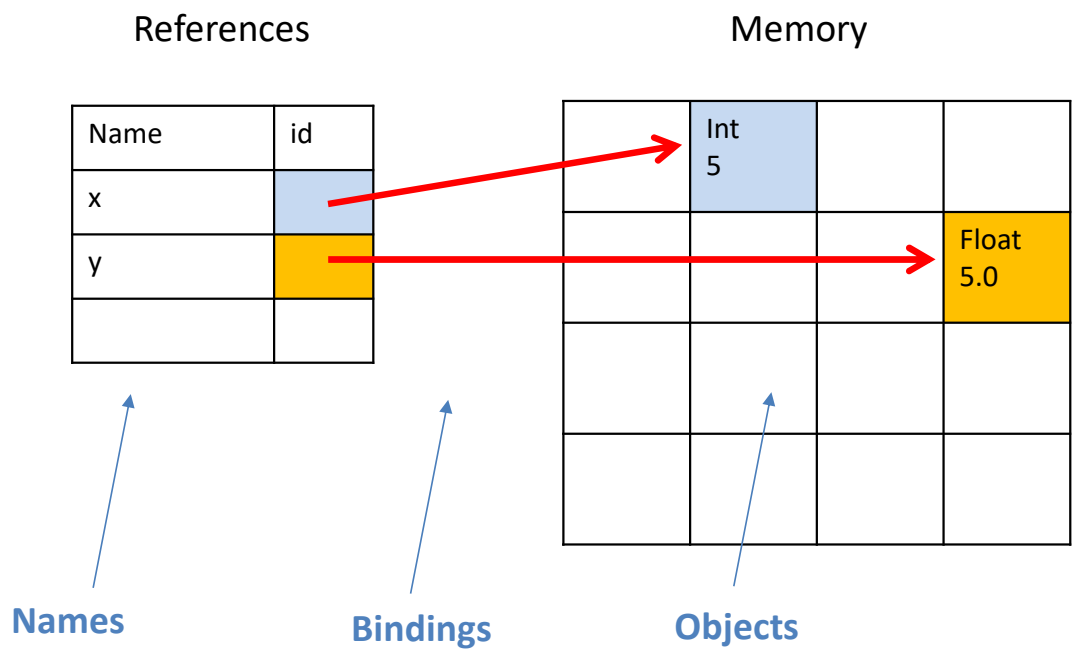
```
>>> x = "my string"
>>> x.upper()
'MY STRING'
```

`dir(`*object*`)` to list all attributes

`help(`*object*`)` to read the "docstring" (documentation string)

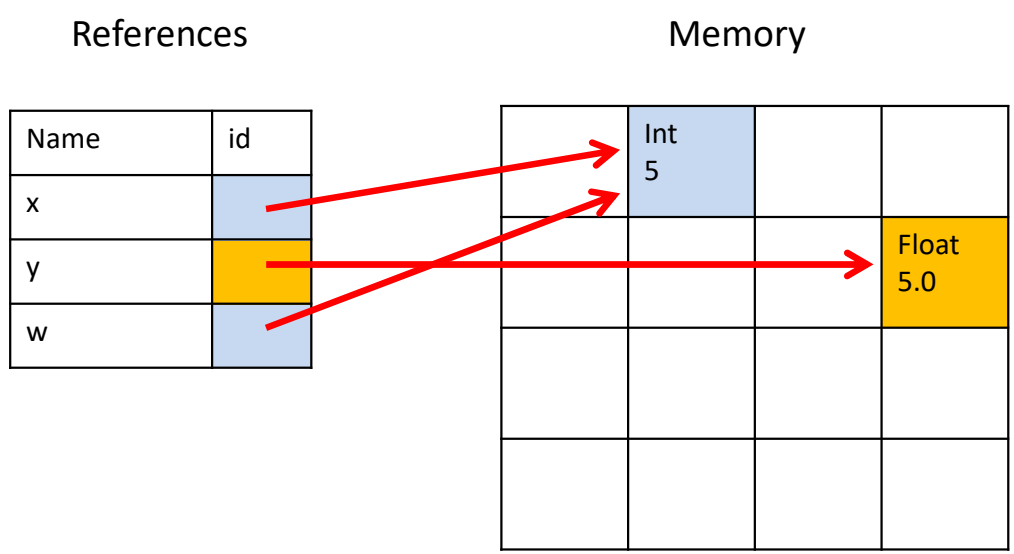
# Python "Variables"

```
x = 5
y = 5.0
w = x
```



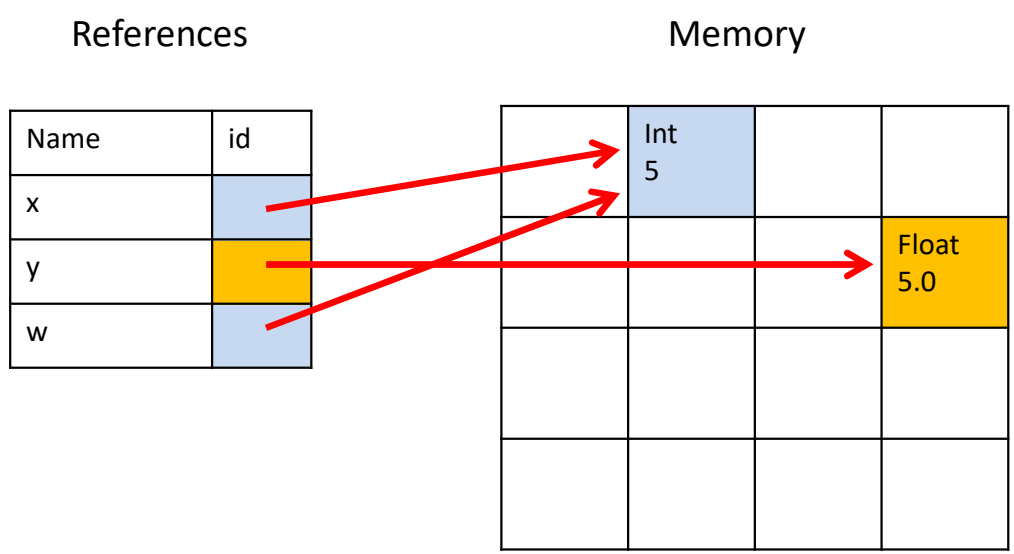
# Python “Variables”

```
x = 5
y = 5.0
w = x
```



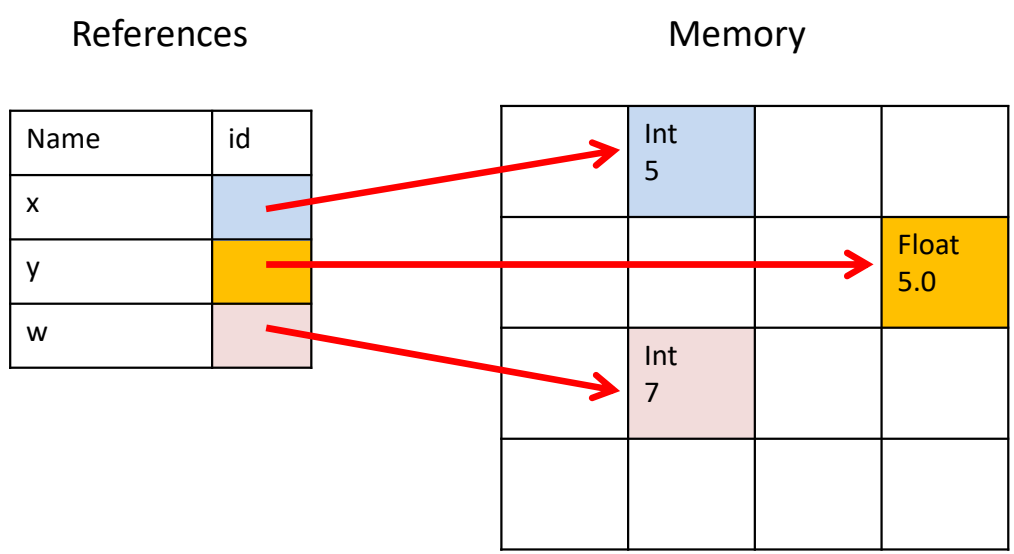
# Python "Variables"

```
x = 5
y = 5.0
w = x
w = x + 2
```



# Python "Variables"

```
x = 5
y = 5.0
w = x
w = x + 2
```



# Variable Declaration

Python is **strongly**, **dynamically** typed

Python uses **"Duck-Typing"**

No declarations, variables can reference any object.

No type checking; objects are defined by their attributes.

# Calling Functions with ()

```
>>> dir
<function dir>

>>> x = dir
>>> x
<function dir>

>>> x()
<results from dir>

>>> dir()
<results from dir>
```

Typing dir without () will return the function itself

This will bind the dir **function** to x

Function called. Will return the **results** of the function rather than the function itself.

# Underscore Patterns

| Pattern | Example | Use |
|---|---|---|
| Double Underscore | __ | placeholder for insignificant values |
| Single Trailing | raise_ | to break naming issues with keywords |
| Single Leading | _myVar | private variables, classes, methods |
| Double Leading | __myVar | fully private variables, classes, methods |
| Double Leading and Trailing | __add__ | reserved for special use in the Python language (never create these) |

"Dunders"

# Example Dunders

| | |
|---|---|
| `__builtins__` | Built-in functions, exceptions and other objects |
| `__cached__` | Name and location of cached files |
| `__file__` | Name and location of the file |
| `__name__` | Name of the module |
| `__doc__` | Docstring: help text for the module |

Operator Methods:

| | |
|---|---|
| `__add__` | When the + operator is used, this dunder is called |
| `__mul__` | When the * operator is used, this dunder is called |

# Demonstrating Example Dunders

Operator Methods:

| | |
|---|---|
| `__add__` | When the + operator is used, this dunder is called |
| `__mul__` | When the * operator is used, this dunder is called |

```
>>> 'b' + 'c'
'bc'
>>> 'b'.__add__(c)
'bc'
```

Calls `str.__add__`

```
>>> "b" + "c"
"bc"
```

Calls `int.__mul__`

```
>>> 2 * 3
6
```

`str.__add__` requires second object to be a string

```
>>> "b" + 5
TypeError
```

# Introspection

| Item | Description |
|---|---|
| dir() | returns list of attributes and methods |
| type() | returns object type |
| id() | returns object unique id |
| is | returns True if both objects are the same |
| sys.getsizeof() | returns number of bytes an objects takes in memory |

# Whitespace

White space is used to define code blocks in Python (such as { } in other languages)

4 spaces or tab

```
>>> x = something
>>> if x > 5:
...     y = x + 2
...     print(y)
>>> print(x)
```

# Special Characters

# comments

\ line continuation

@ decorators

# Numbers

| | |
|---|---|
| int | `y = 5` |
| bool | `y = True` |
| float | `y = 5.0` |
| complex | `y = complex(5,3)`<br>`y = 5 + 3j` |

# Booleans

Every object has a Boolean value

Almost everything is True

Built-in values that are False:

- False
- Any numeric that = 0
- None
- Empty collections: '',[], {}

# Operators

| Arithmetic | |
|---|---|
| Op | Description |
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| ** | Exponent |
| // | Integer division (floor) |
| % | Remainder (modulus) |

```
>>> 5 // 2
2
```

See https://docs.python.org/3/reference/expressions.html#operator-precedence for operator precedence

# Operators

| Comparison | |
|---|---|
| Op | Description |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| == | Equal to |
| != | Not equal to |

```
x < y < z
```

is equivalent to:

```
x < y and y < z
```

```
>>> 5 == 2
False
```

# Operators

| Logical | |
|---|---|
| Op | Description |
| and | logical AND |
| or | logical OR |
| not | logical NOT |

**and, or: Careful when using with non-Booleans!**

```
y = a and b
```
is equivalent to :
```
if bool(a):
    y = b
else:
    y = a
```

```
y = a or b
```
is equivalent to :
```
if bool(a):
    y = a
else:
    y = b
```

```
>>> False and my_func()
(not evaluated)

>>> True or my_func()
(not evaluated)
```

**not: Always returns a Boolean**

```
>>> not(5)
False
```

# Operators

| Identity | |
|---|---|
| Op | Description |
| is | same object |
| is not | not same object |

## is

Returns `True` if both operands have same id

```
>>> a = 257
>>> b = a

>>> a is b
```

result is `True`

```
>>> a = [257]
>>> b = [257]

>>> a is b
```

result is `False`

Will come back to this with Mutable / Immutable

# Operators

| Membership | |
|---|---|
| Op | Description |
| in | variable is in collection |
| not in | variable is not in collection |

## in

Returns `True` if variable is in a collection

```
a = [1, 3, 5]
3 in a
```

result is `True`

```
3.0 in a
```

result is `True`

# Operators

| Bitwise | |
|---|---|
| Op | Description |
| >> | right shift |
| << | left shift |
| & | bitwise AND |
| ^ | bitwise XOR |
| \| | bitwise OR |
| ~ | bitwise NOT |

```
255 >> 2
```

result is 63

See https://docs.python.org/3/reference/expressions.html?highlight=subscriptions#grammar-token-subscription

# Strings

Can use single or double quotes

```
>>> x = 'my string'
>>> x = "my string"
>>> x = 'this is what I "would" do'
```

Triple quotes  to keep newlines and whitespace

```
>>> x = """this is a long multiline

       string with whitespace and tabs"""
```

See PEP257 for docstring conventions:
https://www.python.org/dev/peps/pep-0257/

# Strings

Escape sequences:

| | | | |
|---|---|---|---|
| `\` | | | |
| `\\` | back slash | | |
| `\'` | single quote | | |
| `\"` | double quote | | |
| `\n` | linefeed | | |
| `\t` | tab | | |
| `\x`*hh* | hex character | `print("\x24")` | `$` |
| `\`*ooo* | octal character | `print("\044")` | `$` |

preface string with r to have backslash literals:

```
>>> r"these \\ backslashes \ are literal"
```

preface string with b to have byte characters

```
>>> b'$' == b'\x24' == b'\044'
```

# Formatting Strings

```
>>> w = 11
>>> k = 13

>>> f"string with {w} and {k} inserted"
```
returns: string with 11 and 13 inserted

```
>>> f"string with {w:x} and {k:b} inserted"
```
returns: string with b and 1101 inserted

```
>>> f"string with {w+5:10.3f} inserted"
```
returns: string with  16.000 inserted

see https://pyformat.info/ for new f-string and older (.format{} and %) formatting styles

see https://docs.python.org/3/reference/lexical_analysis.html#f-strings for all options

# Mutable / Immutable

**mutable** objects **can be changed** at current memory assignment ("in-place")

**immutable** objects **cannot be changed**, a new object will be created

**Key point:** Immutable objects are quicker to access, but expensive if a change is needed as it will need to be recreated

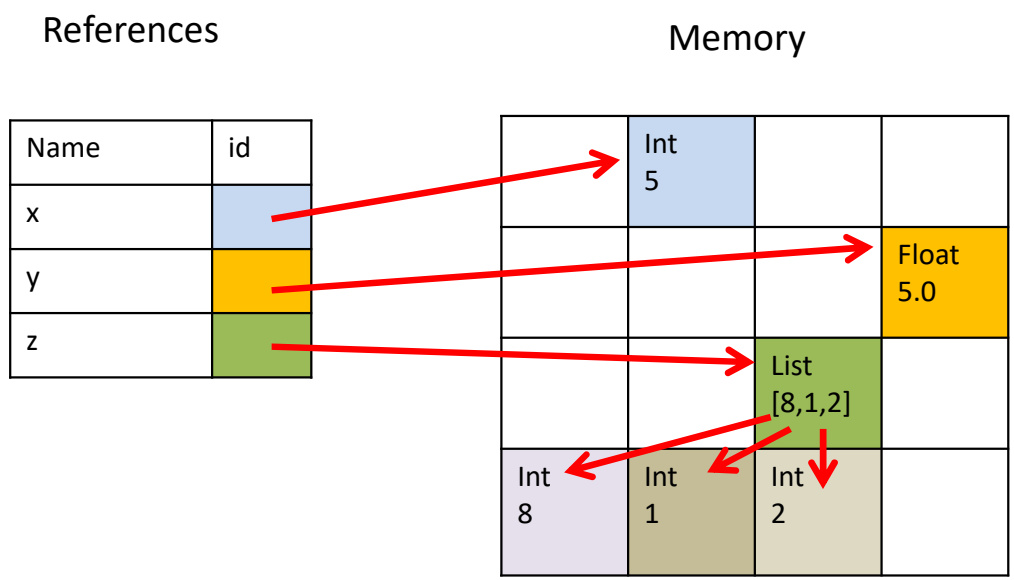
# Mutable / Immutable

| Mutable | Immutable |
|---|---|
| list | bool |
| set | int, float, complex |
| dict | tuple |
| byte arrays | str |
| | frozenset |

Objects that we typically change the size of

# Mutable / Immutable

```
x = 5
y = 5.0
z = [8,1,2]
```



# Mutable Example (1)

```
x = [-7,5.0]
```

References

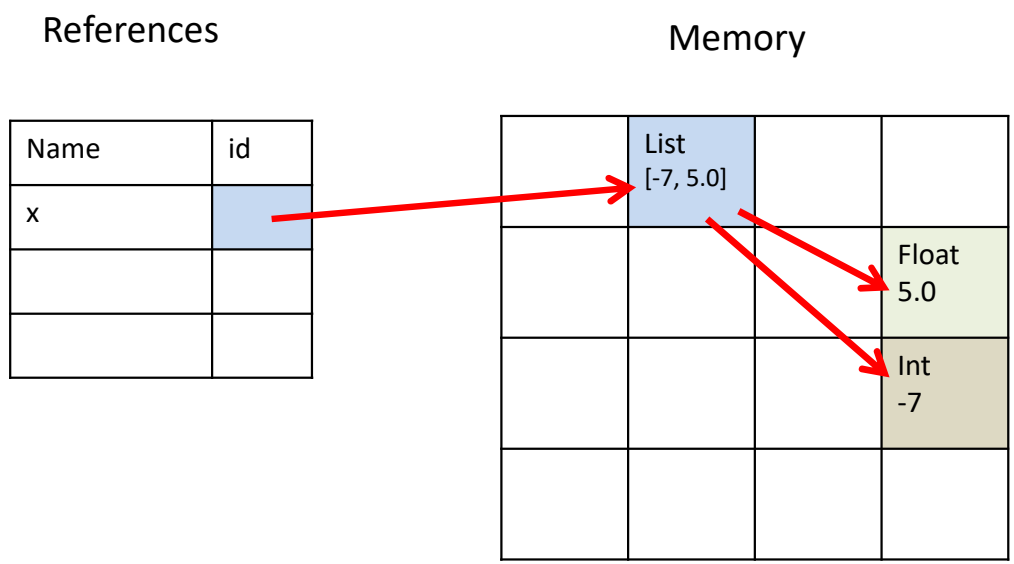
| Name | id |
|---|---|
| | |
| | |
| | |

Memory

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

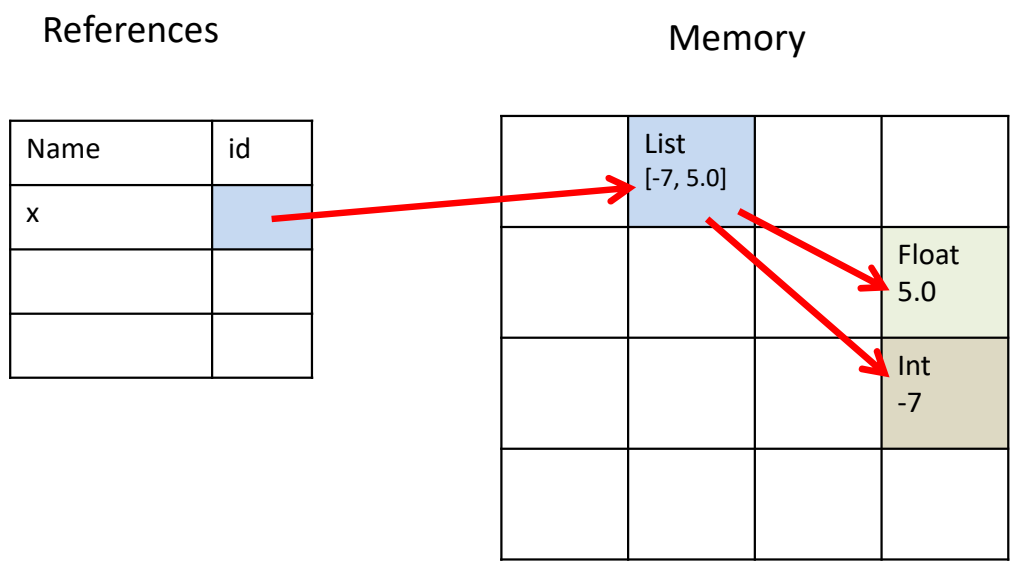
# Mutable Example (1)

```
x = [-7,5.0]
```



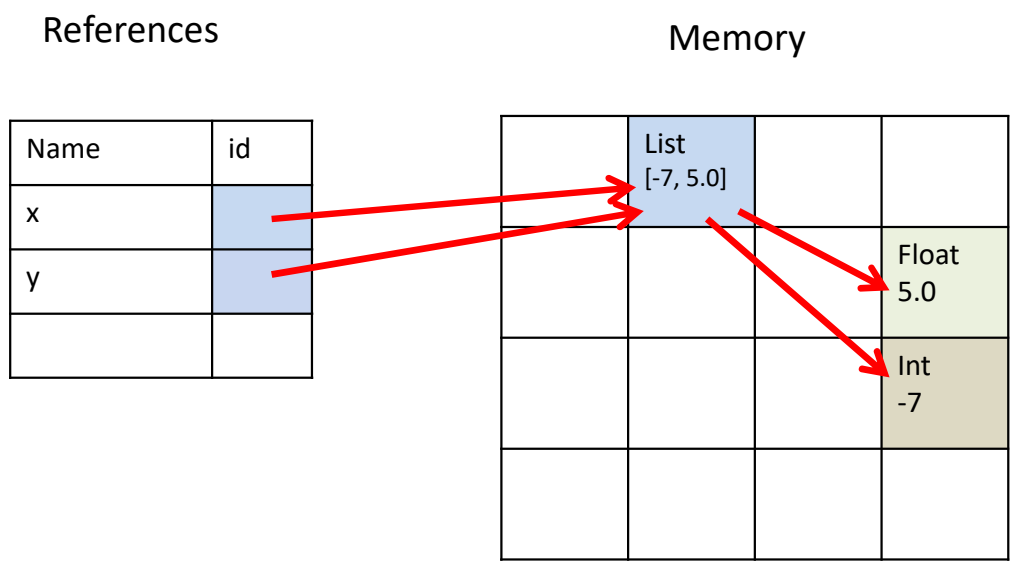
# Mutable Example (1)

```
x = [-7,5.0]
y = x
```



# Mutable Example (1)

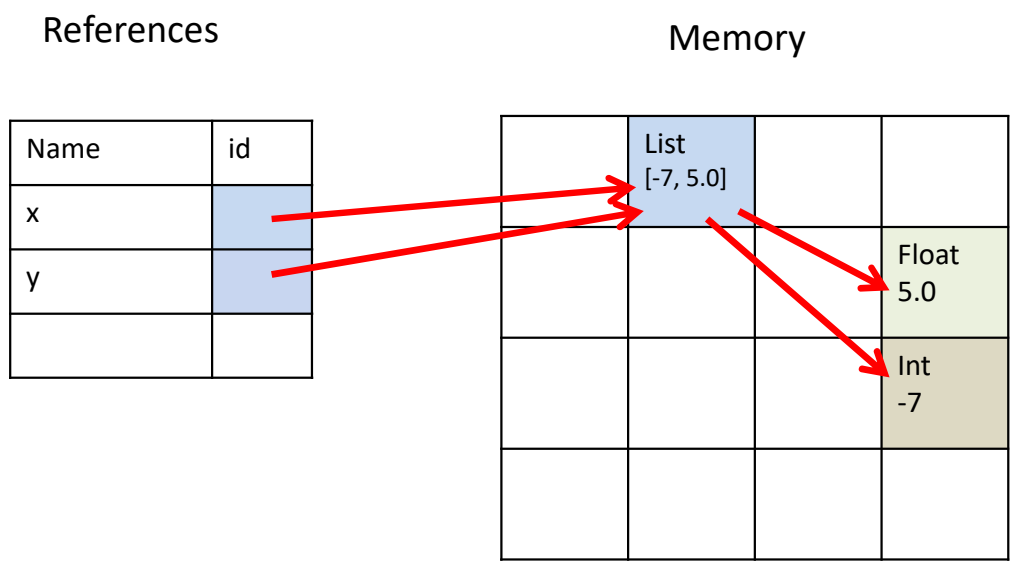
```
x = [-7,5.0]
y = x
```




```
>>> x is y
True
```

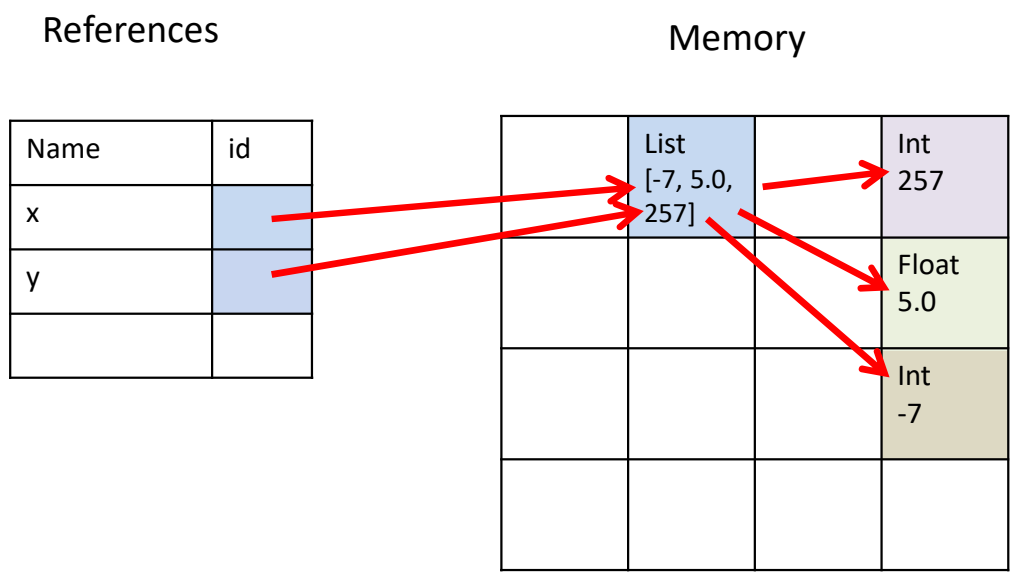
# Mutable Example (1)

```
x = [-7,5.0]
y = x
x.append(257)
```



# Mutable Example (1)

```
x = [-7,5.0]
y = x
x.append(257)
```




```
>>> print(x)
[-7, 5.0, 257]
>>> print(y)
[-7, 5.0, 257]
```

# Mutable Example (2)

```
x = [-7,5.0]
```

References

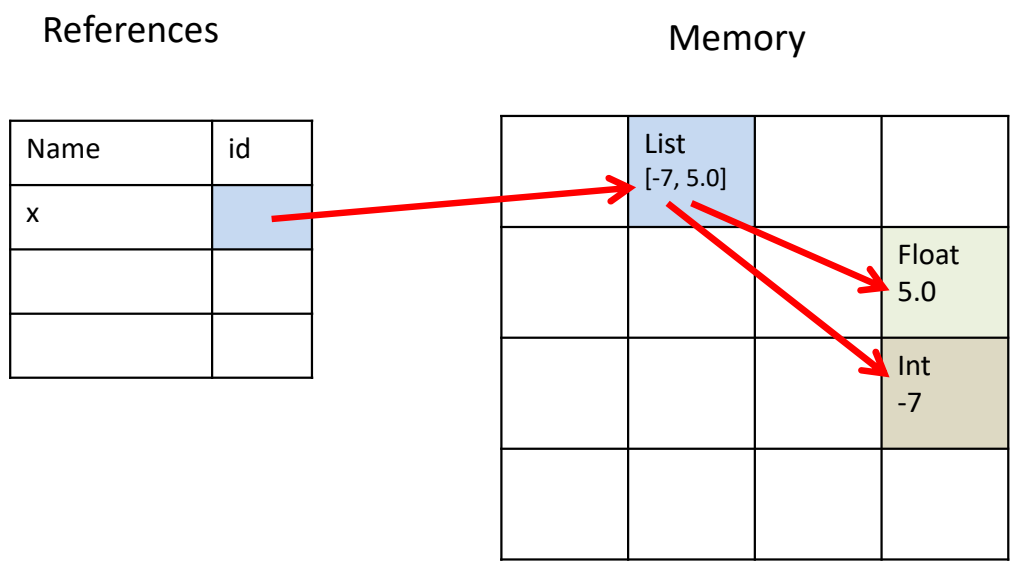
| Name | id |
|---|---|
| | |
| | |
| | |

Memory

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

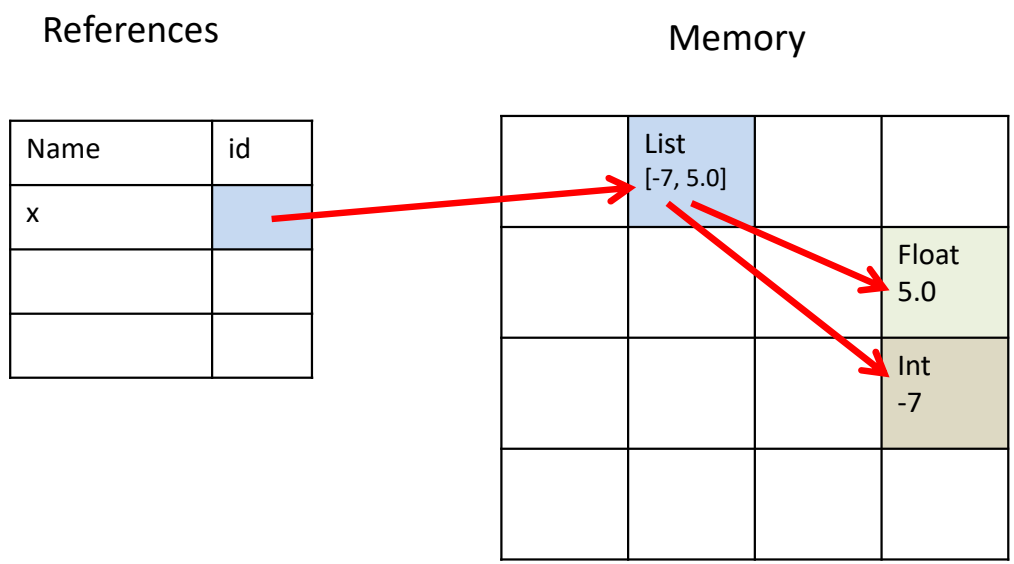
# Mutable Example (2)

```
x = [-7,5.0]
```



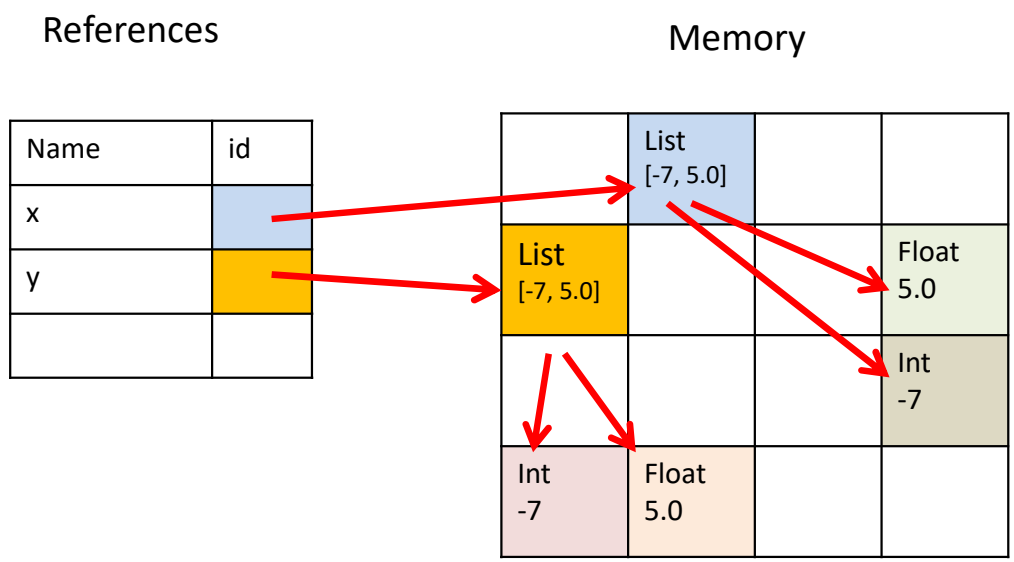
# Mutable Example (2)

```
x = [-7,5.0]
y = [-7,5.0]
```



# Mutable Example (2)

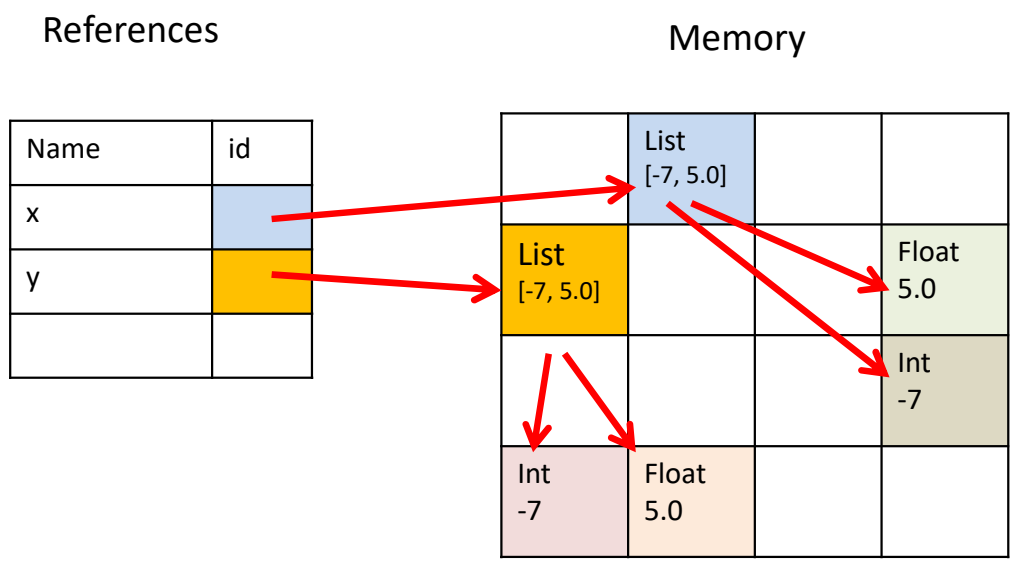
```
x = [-7,5.0]
y = [-7,5.0]
```




```
>>> x is y
False
```

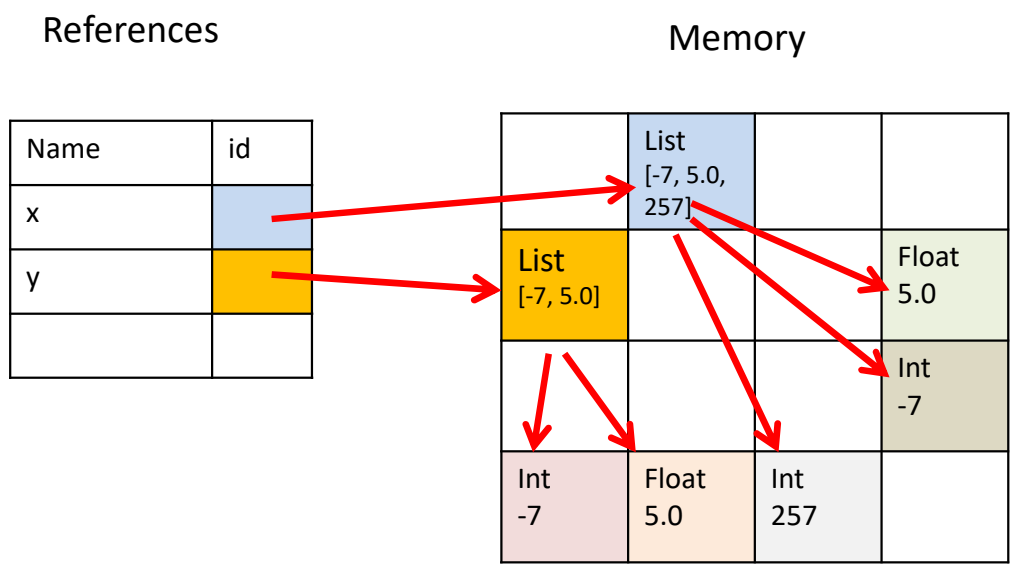
# Mutable Example (2)

```
x = [-7,5.0]
y = [-7,5.0]
x.append(257)
```



# Mutable Example (2)

```
x = [-7,5.0]
y = [-7,5.0]
x.append(257)
```




```
>>> print(x)
[-7, 5.0, 257]
>>> print(y)
[-7, 5.0]
```

# Immutable Example (1)

```
x = 257
```

References

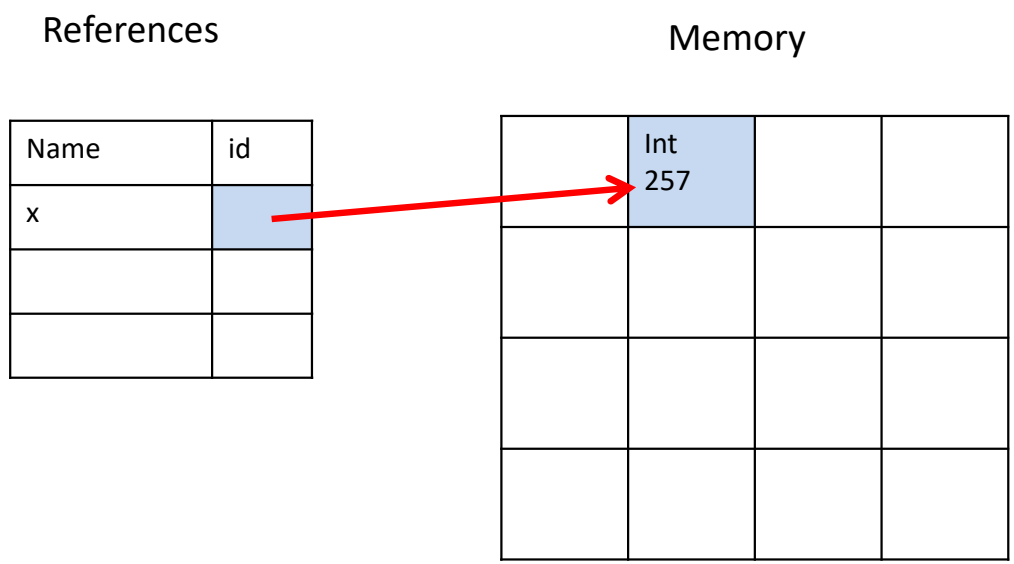
| Name | id |
|---|---|
| | |
| | |
| | |

Memory

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

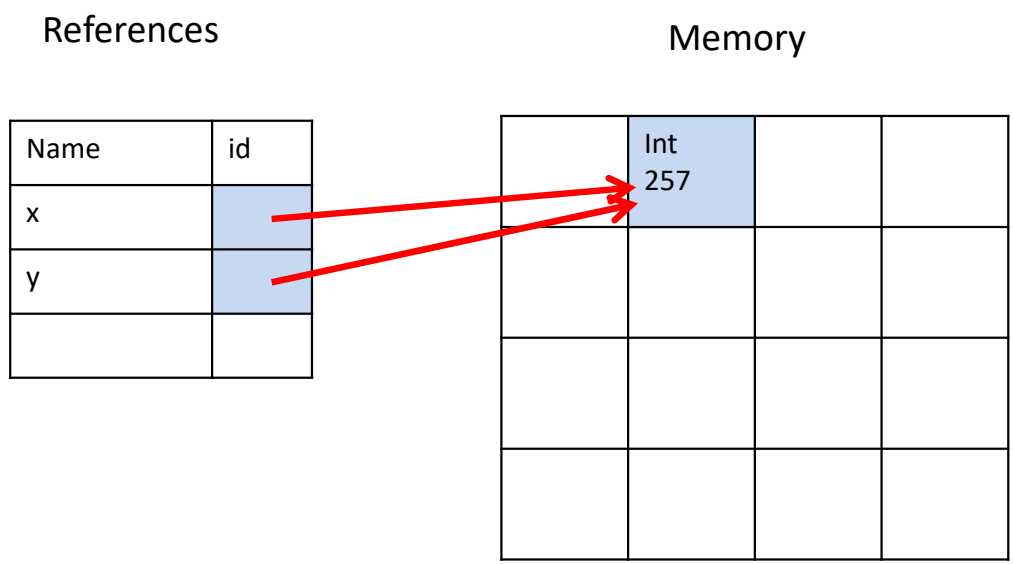
# Immutable Example (1)

```
x = 257
y = x
```



# Immutable Example (1)

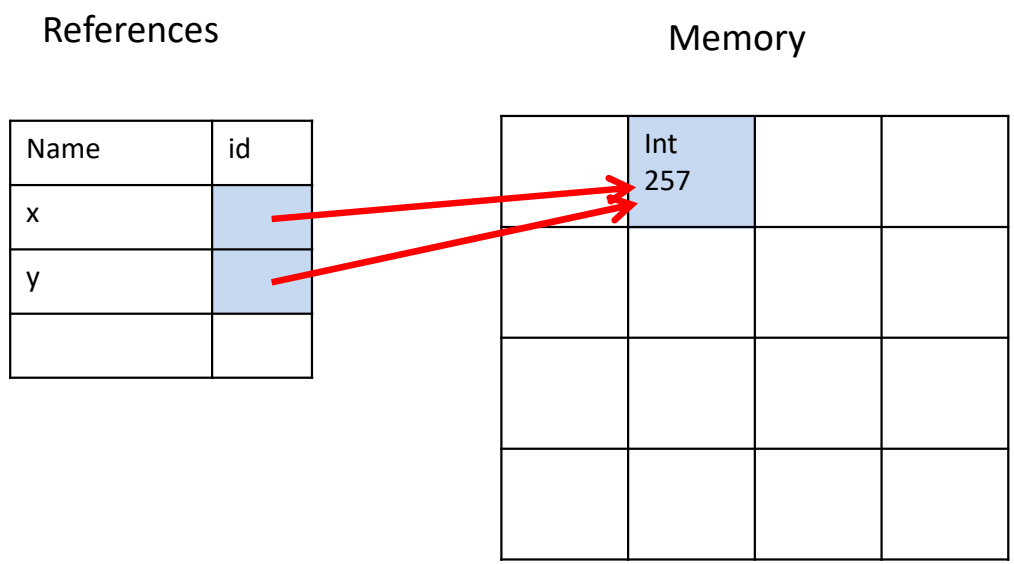
```
x = 257
y = x
```




```
>>> x is y
True
```

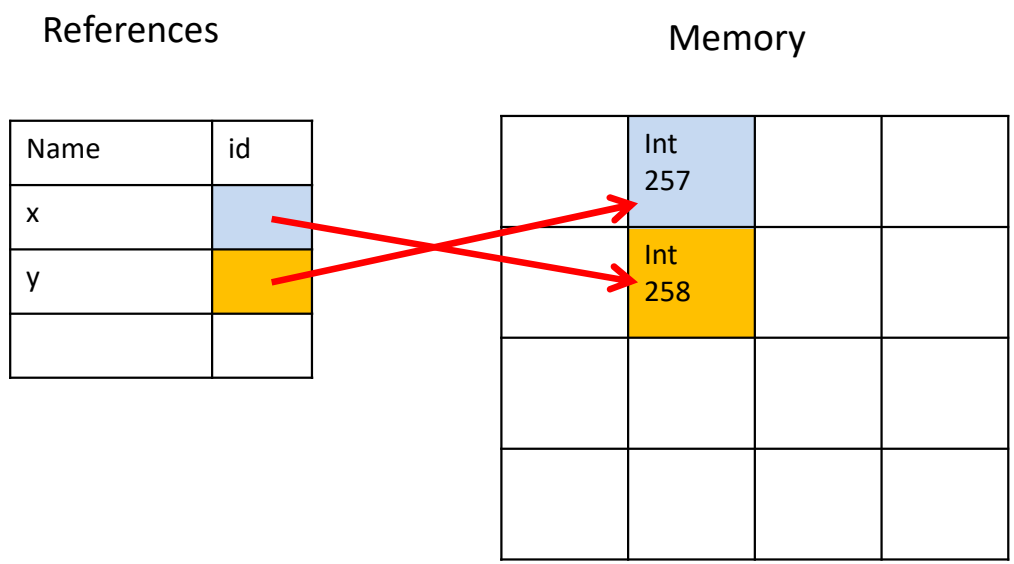
# Immutable Example (1)

```
x = 257
y = x
x = 258
```



# Immutable Example (1)

```
x = 257
y = x
x = 258
```




```
>>> x is y
False
```

# Immutable Example (2)

```
x = 257
```

References

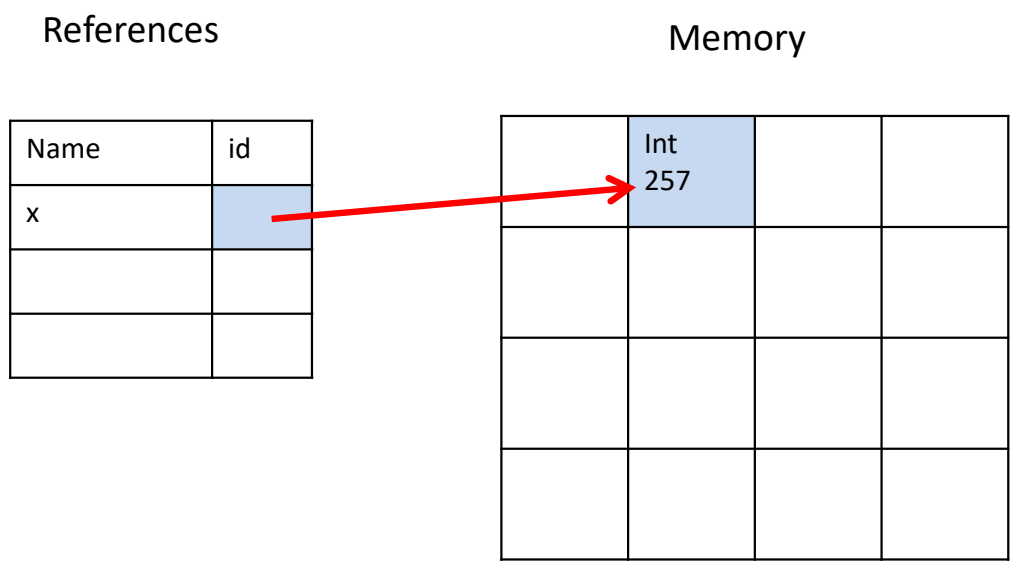
| Name | id |
|---|---|
| | |
| | |
| | |

Memory

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

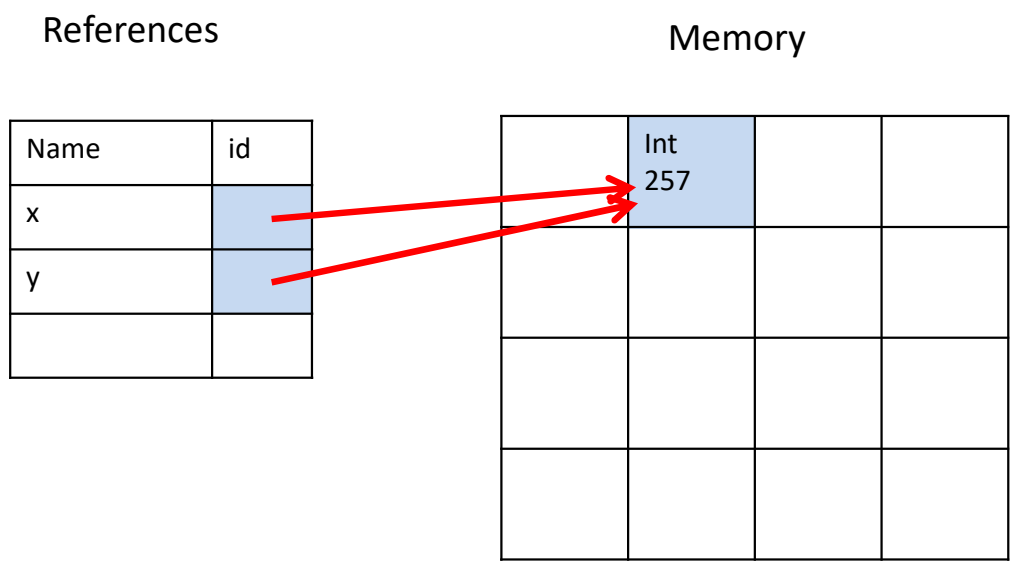
# Immutable Example (2)

```
x = 257
y = x
```



# Immutable Example (2)

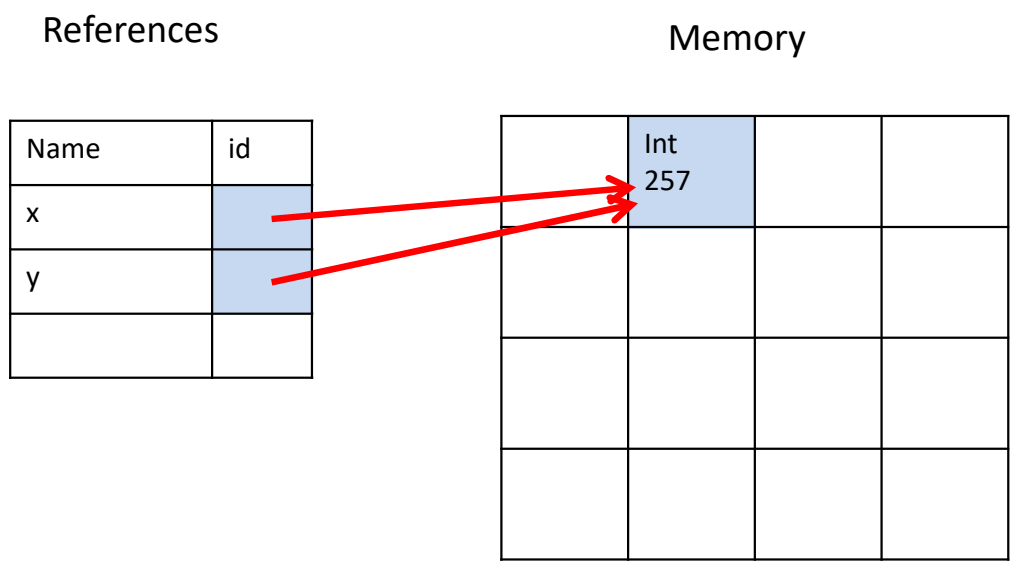
```
x = 257
y = x
```




```
>>> x is y
True
```

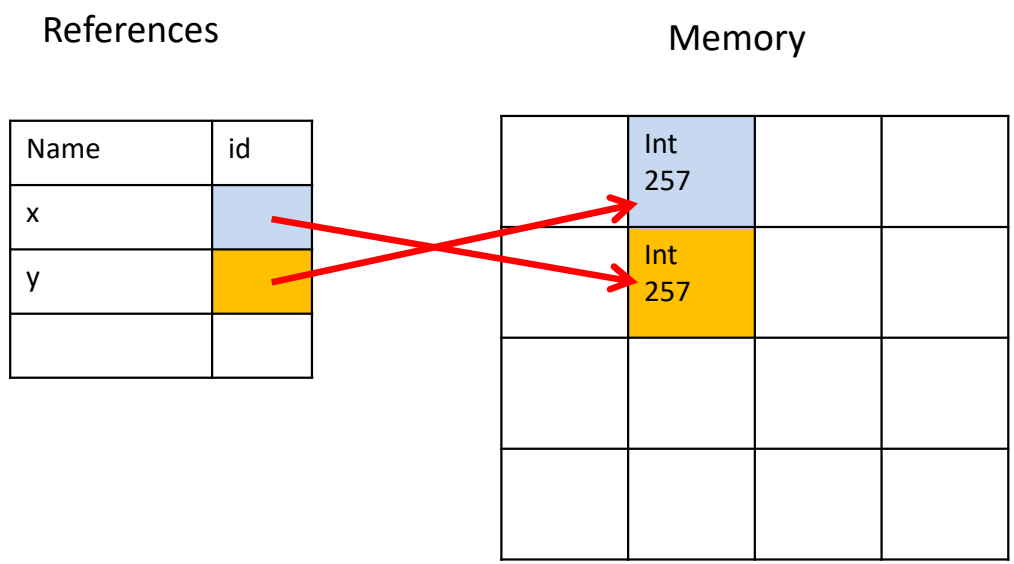
# Immutable Example (2)

```
x = 257
y = x
x = 257
```



# Immutable Example (2)

```
x = 257
y = x
x = 257
```




```
>>> x is y
False
```

# Additional Resources

Useful Python snippets
https://www.pythonsheets.com/

Python documentation on the standard library
https://docs.python.org/3/library/

Python Style Guide
https://www.python.org/dev/peps/pep-0008/

# HW Assignment

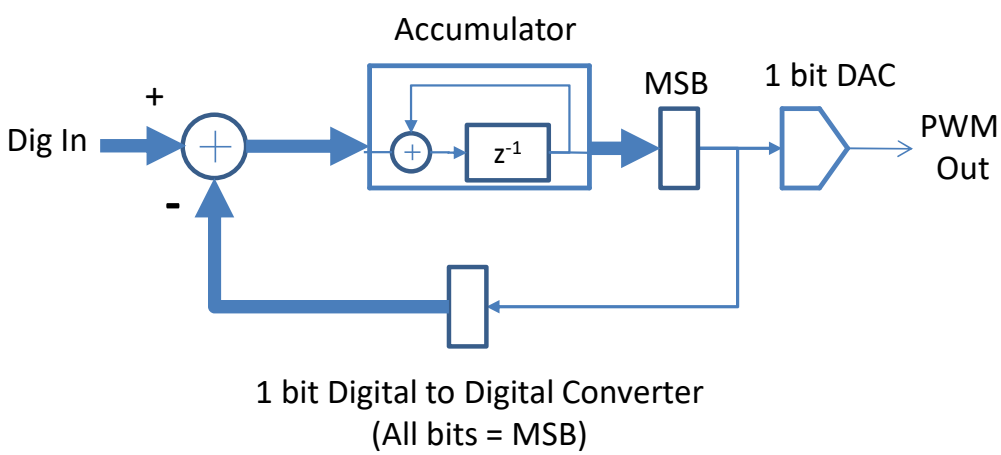
This will be detailed in the next Workshop Q/A session. Please try to get as far as you can on your own before the Workshop.

Goal:

Using Spyder, create a script of a simple first-order Delta Sigma DAC per the function definition prototype and block diagram on next slides.

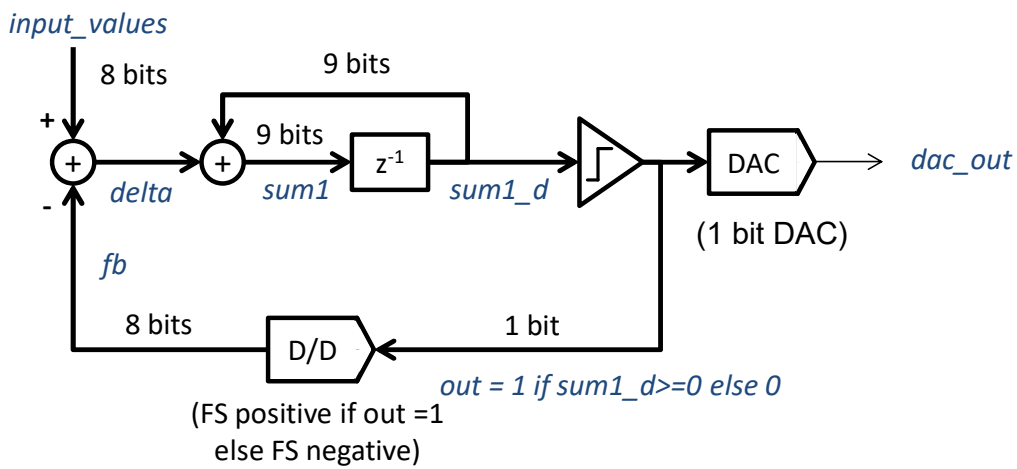
Refer back to Simple Coding Example and overview of Spyder from Session 1 to get started.

# 1st Order Delta Sigma DAC



# DAC Implementation

All *values* are represented as signed integers from $-2^{b-1}$ to $+2^{b-1}-1$



# For HW: Lists, For Loop, If statement

Creating a list of input values : `input_values = [-20] * N` (The * operator for lists will replicate the list contents N times)

Could have a list of time varying inputs: `input_values = [-20, -19, -18, -17, -16]`
(A lot of typing now, we'll learn how to create these from functions later so for now just use the *N approach representing samples of a constant input)

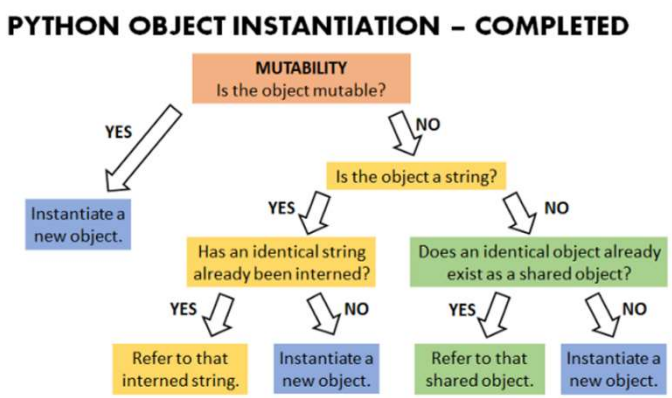
Create an empty list to store the output values: `dac_out = []`

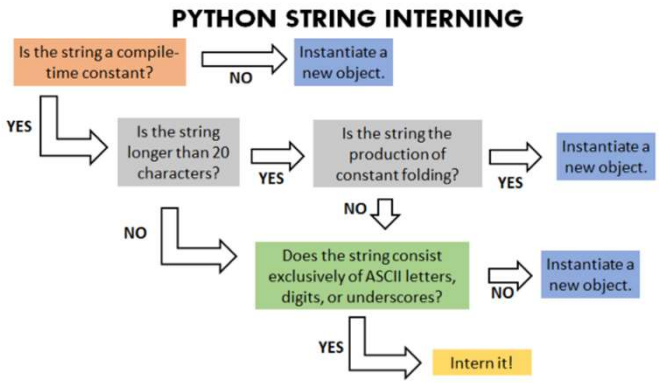
Iterate through input_values in a for loop and use a list append method to add outputs after computation in each loop cycle

```
# assigns each item in input_values to the name "sample" in each iteration
for sample in input_values:
    # (compute internal states for each clock cycle, synchronous items first)
    ...
    # (conditional tests):
    if (test):
          # stuff to do if test is true
    else:
          # stuff to do if test is false
   # update output:
   dac_out.append(new_output_value)
```

# BACK-UP SLIDES

# Object Instantiation




This is an interesting detail of CPython (which we are using) that is simply an interpreter optimization and is not at all critical to understand. This explains some of the behavior you may see if you dig in further (by introspection using id(x) and **b is y)** to shared references. Interning and caching is why **x=5** and **y =5** will point to the same object but **x=259** and **y=259** will not (integers in the range of -5 to 256 are cached). Many strings as described on this slide are similarly interned for reuse. Regardless of this happening, it does NOT change the high level behavior regarding mutability/immutability: You cannot use the assignment operator (=) to change an object, only it's mutable methods. Even if these immutable objects (integers and strings) have a shared reference, changing one will always create a new object and not affect the other.

Source: https://medium.com/@bdov_/https-medium-com-bdov-python-objects-part-iii-string-interning-625d3c7319de

# Expressions vs Statements

**Statements: an instruction the Python interpreter can execute**

```
>>> x = 5
>>> print("this")
```

**Expressions: a combination identifiers, literals, operators and calls to functions and reduced to some kind of value**

Identifiers: name used to identify a variable
Literals: built-in objects such as string, integer, float, long, list, tuple, etc
Operators: function and subscription operators (), [], ternary operator

Expressions need to be evaluated, and result is a value