DATA TYPES

Overview:

- what are the different data types in Python
- primitive vs. collection types
- how to construct data types
- numeric types and their representation
- numeric and modulo arithmetic
- examine and contrast operations (numeric, bitwise, comparison, logical & identity)
- membership and object comparison

Python Data Types

- building blocks in a language
- similar to noun, verb
- Python has two groups of types
  1. primitive types ("*atoms*")
  2. collections ("*molecules*")
- additional special types:
  1. *None* type
  2. *range* type

Operators

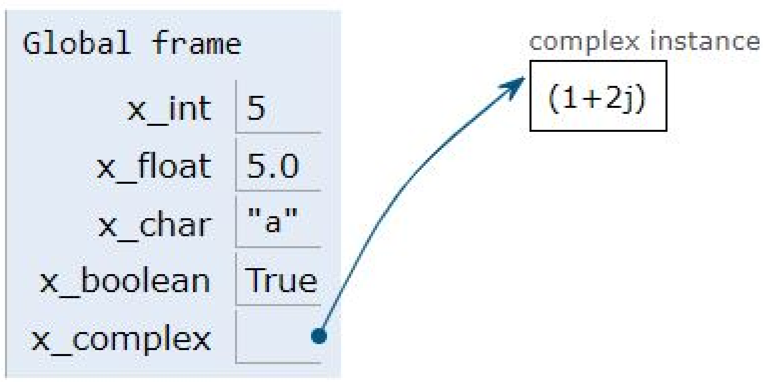
- arithmetic: $+, -, *, **, /, //, **$
- assignment:

  $$=, +=, -=, *=, /=, \%=, //=$$

- bitwise: $\&, |, \hat{}\ , <<, >>$
- comparison: $==, !=, <, <=, >, >=$
- logical: *and*, *or*, *not*
- identity: *is*, *is not*
- membership: *in*, *not in*
- many are *polymorphic*

Primitive Types

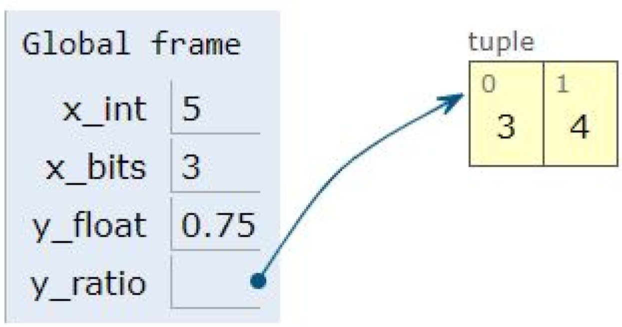
```
x_int     = 5
x_float   = 5.0
x_char    = 'a'
x_boolean = True
x_complex = 1 + 2j
```



- "*atoms*" - indivisible objects

Primitive Type Examples

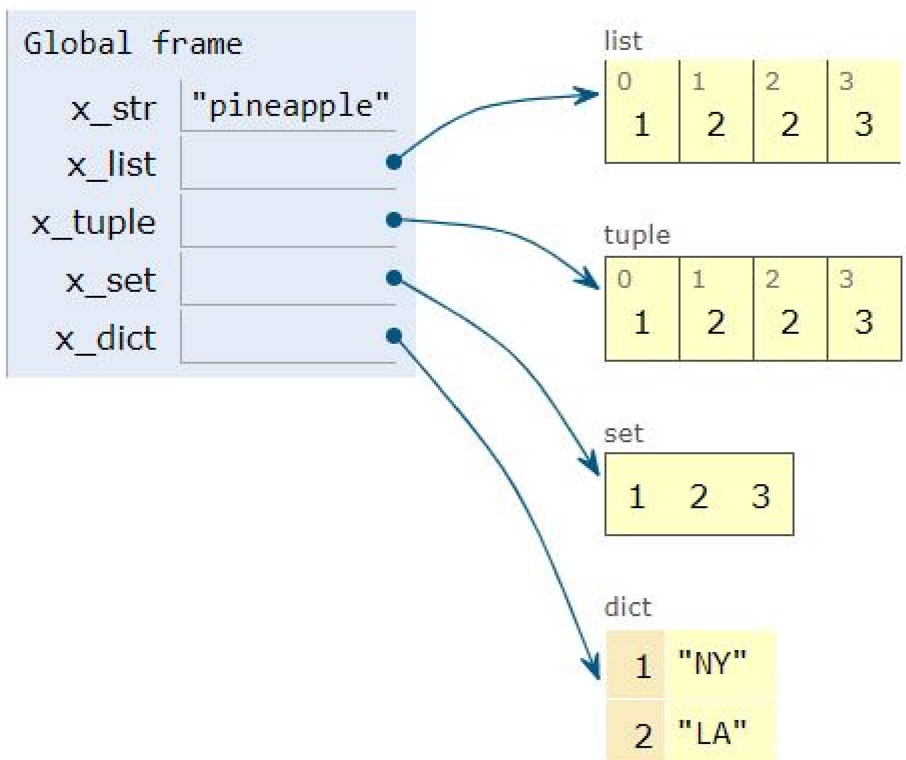
```
x_int   = 5
x_bits  = x_int.bit_length()

y_float = 0.75
y_ratio = y_float.as_integer_ratio()
```



- ”*atoms*” are not just values
- objects with methods

Collection Types

```
x_str     = 'pineapple'
x_list    = [1,  2, 2, 3]
x_tuple   = (1, 2, 2, 3)
x_set     = {1, 2, 2, 3} # note duplicates
x_dict    = {1: 'NY', 2: 'LA'}
```



- "*molecules*" - complex ob-

jects

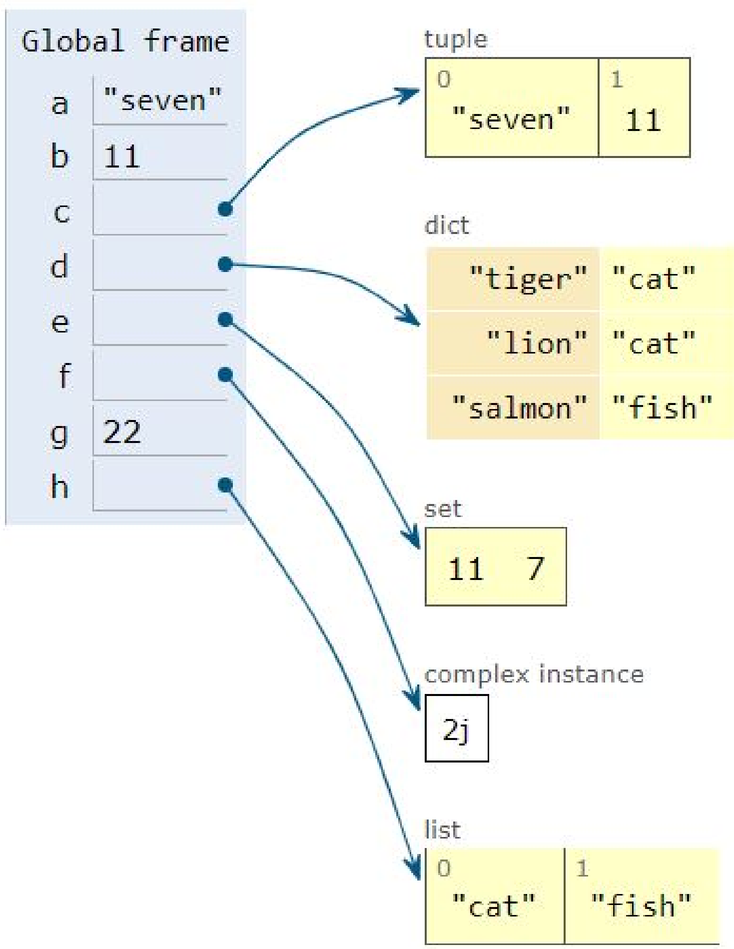
Constructors for Types

```
x_str     = 'pineapple'
y_str     = str('pineapple')

x_list    = [1,  2, 2, 3]
y_list    = list((1, 2, 2, 3))

x_tuple   = (1, 2, 2, 3)
y_tuple   = tuple((1,2,2,3))

x_set     = {1, 2, 2, 3}
y_set     = set((1,2,2,3))

x_dict    = {1: 'NY', 2: 'LA'}
y_dict    = dict({1: 'NY', 2: 'LA'})
```

- note: double brackets

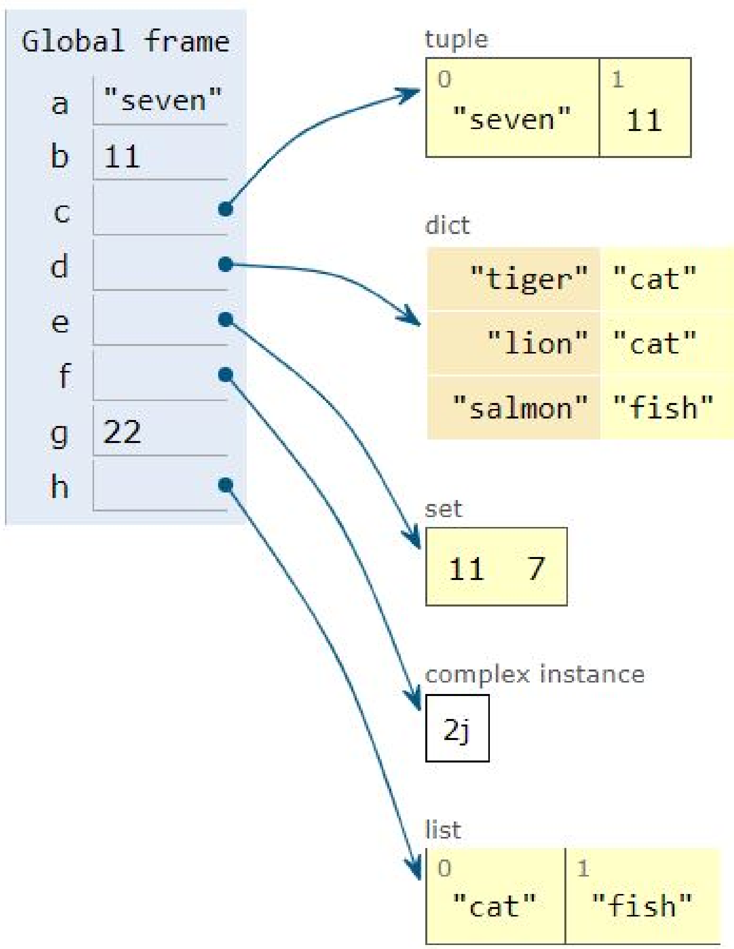
Exercise(s):

- identify primitive and collection types



Exercise(s):

- write Python code to define objects in the picture

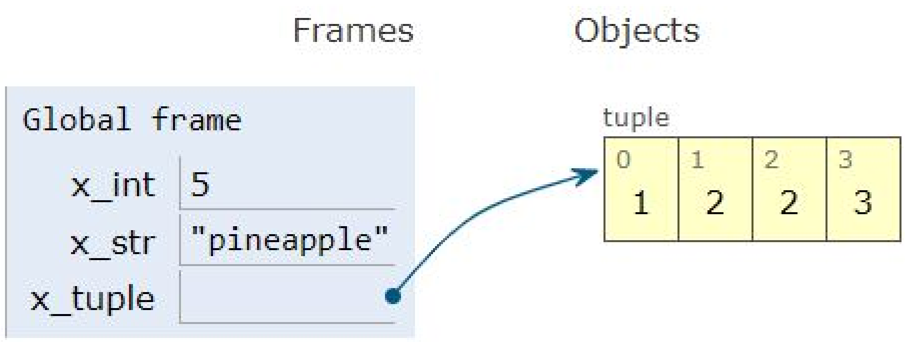


type() Function

```python
x_int    = 5
x_str    = 'pineapple'
x_tuple  = (1, 2, 2, 3)
print(x_int, type(x_int))
print(x_str, type(x_str))
print(x_tuple, type(x_tuple))
```

Print output (drag lower right corner to resize)

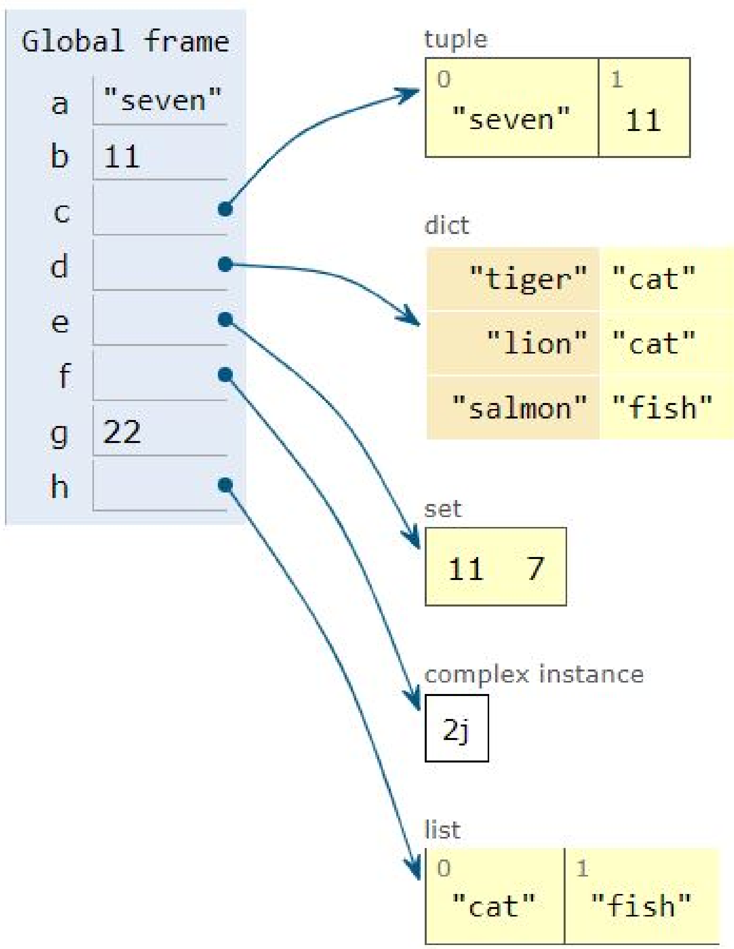
```
5 <class 'int'>
pineapple <class 'str'>
(1, 2, 2, 3) <class 'tuple'>
```



- *type*() is *polymorphic*

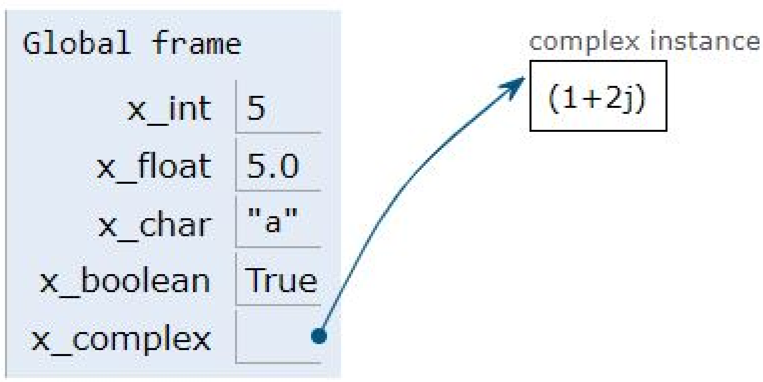
Exercise(s):

- write code to print type of each object shown below



Numeric Types

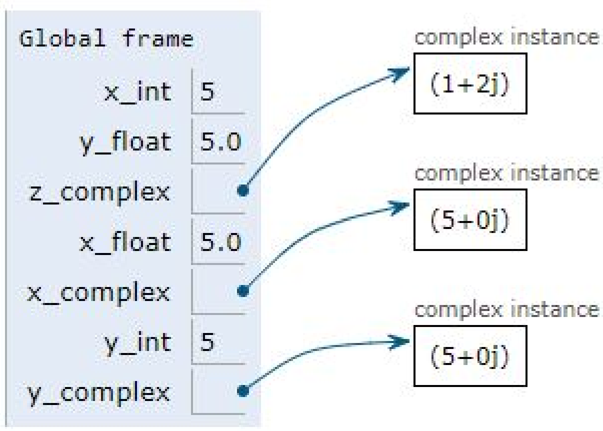
```python
x_int     = 5
x_float   = 5.0
x_char    = 'a'
x_boolean = True
x_complex = 1 + 2j  # same as complex(1, 2)
```



- three numeric types:
  integer, float (real), complex

int(), *float*(), *complex*()

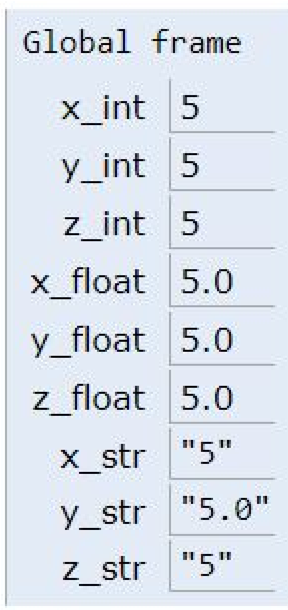
```
x_int     = 5
y_float   = 5.0
z_complex = 1 + 2j

x_float   = float(x_int)
x_complex = complex(x_int)
y_int     = int(y_float)
y_complex = complex(y_float)
```



- type ”casting”

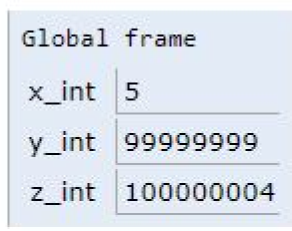
Type Casting

```
x_int=int(5); y_int=int(5.0); z_int=int("5")

x_float = float(5);     y_float = float(5.0)
z_float = float('5')

x_str=str(5); y_str=str(5.0); z_str=str("5")
```



Integer Type

```
x_int = 5 # numeric literal
y_int = 999999999999999999999999
z_int = x_int + y_int
```



- unlimited length
- integers are objects with methods

Integer Representation

- different bases: 2, 8, 10, 16

```
x_int = 30   # default: base 10
x_bin = 0b11110       # binary literal
x_oct = 0o36          # octal literal
x_hex = 0x1E          # hex literal
```

Global frame

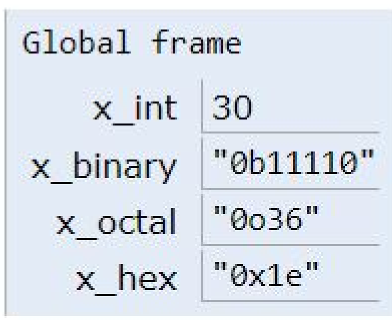
x_int	30
x_bin	30
x_oct	30
x_hex	30

$$\text{x\_int} = 3 \cdot 10^1 + 0 \cdot 10^0$$
$$\text{x\_bin} = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$
$$\text{x\_oct} = 3 \cdot 8^1 + 6 \cdot 8^0$$
$$\text{x\_hex} = 1 \cdot 16^1 + 14 \cdot 16^0$$

Integer Conversion

- different bases: 2, 8, 10, 16

```
x_int = 30   # default: base 10
x_bin = bin(x_int)   # binary: base 2
x_oct = oct(x_int)   # octal: base 8
x_hex = hex(x_int)   # hex:   base 16
```



$$\text{x\_int} = 3 \cdot 10^1 + 0 \cdot 10^0$$
$$\text{x\_bin} = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$
$$\text{x\_oct} = 3 \cdot 8^1 + 6 \cdot 8^0$$
$$\text{x\_hex} = 1 \cdot 16^1 + 14 \cdot 16^0$$

Exercise:

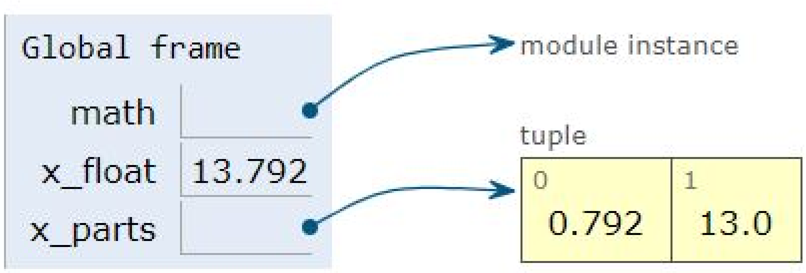
- represent each decimal number in base 2, 8 and 16:

(a) 21
(b) 39
(c) 64
(d) 96
(e) 100

Floating Point Numbers

- real numbers
- integer and fractional part
- 64-bit double precision

```
import math
x_float = 13.792
x_parts = math.modf(x_float)
```




```
y_float = 2.86e2      # scientific notation
```

Complex Numbers

$$x^2 - 2x + 10 = 0$$

- (conjugate) complex roots:

  $c_1 = 1-4i,\ c_2 = 1+4i,\ i = \sqrt{-1}$

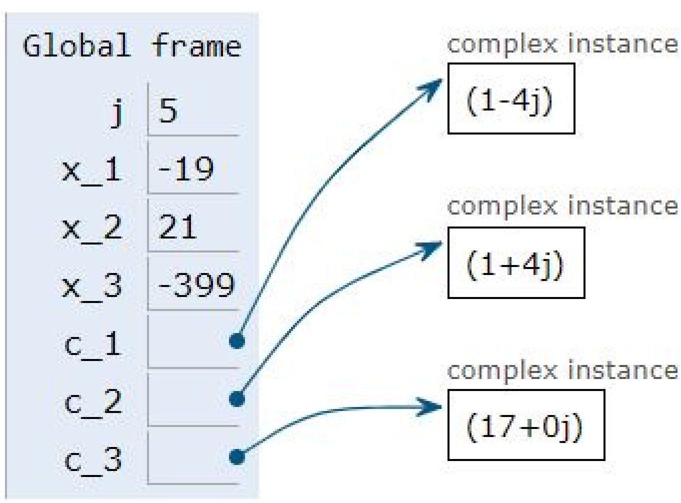
- Python uses j for $\sqrt{-1}$

```
c_1 = 1-4j
c_2 = 1 + 4j
```

- note: $1 + 4j \neq 1 + 4 * j$

Complex Numbers (cont'd)

```
j   = 5
x_1 = 1 - 4 * j
x_2 = 1 + 4 * j
x_3 = x_1 * x_2
c_1 = 1 - 4j     # no spaces before j
c_2 = 1 + 4j     # 1 + 4 j is illegal
c_3 = c_1 * c_2
```



Numeric Operations

`+, -, *, **, /, %, //, **`

- operator precedence

1. ()
2. $**$ (exponentiation)
3. $-x$ negation
4. $*$, $/$, %, $//$
5. $+$, $-$

- same precedence: left to right

Numeric Operations (cont'd)

```
x = -1**2;  x_1 = (-1)**2;    x_2 = -(1**2)

y = 2+2*2;  y_1 = (2+2)*2;    y_2 = 2+(2*2)

z = 2*2**2; z_1 = (2*2) **2; z_2 = 2*(2**2)
```

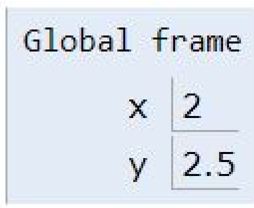
Global frame

x_1	1
x_2	-1
x	-1
y_1	8
y_2	6
y	6
z_1	16
z_2	8
z	8

Division in Python

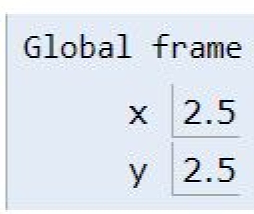
- division in Python 2.7

```
x = 5/2
y = 5.0/2
```



- division in Python 3.6

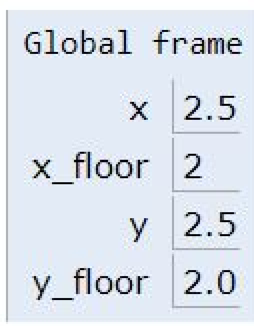
```
x = 5/2
y = 5.0/2
```



Floor Division //

- returns the quotient

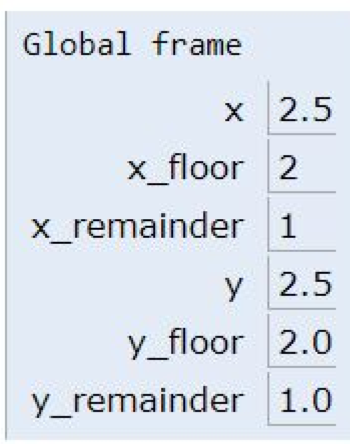
```
x       = 5 /  2
x_floor = 5 // 2
y       = 5.0 / 2
y_floor = 5.0 // 2
```



Modulo Arithmetic %

- returns the remainder

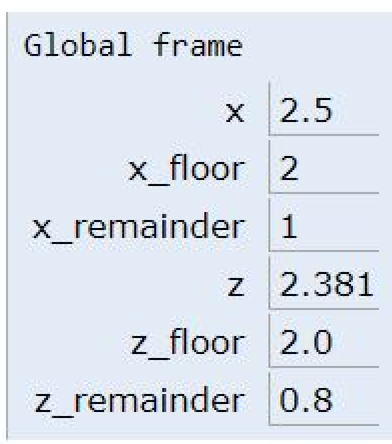
```
x           = 5 /  2
x_floor     = 5 // 2
x_remainder = 5 % 2

y           = 5.0 / 2
y_floor     = 5.0 // 2
y_remainder  = 5.0 % 2
```



Modulo Arithmetic % (cont'd)

- can use fractional modulus

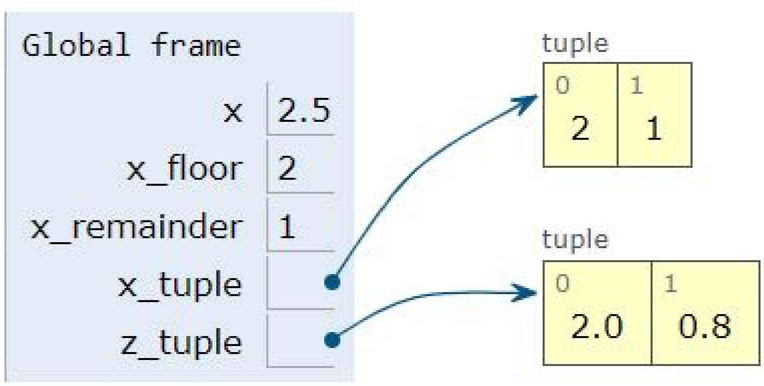
```
x           = 5 /  2
x_floor     = 5 // 2
x_remainder = 5 % 2

z           = 5.0 / 2.1
z_floor     = 5.0 // 2.1
z_remainder  = 5.0 % 2.1
```



divmod() Function

- combines // and %

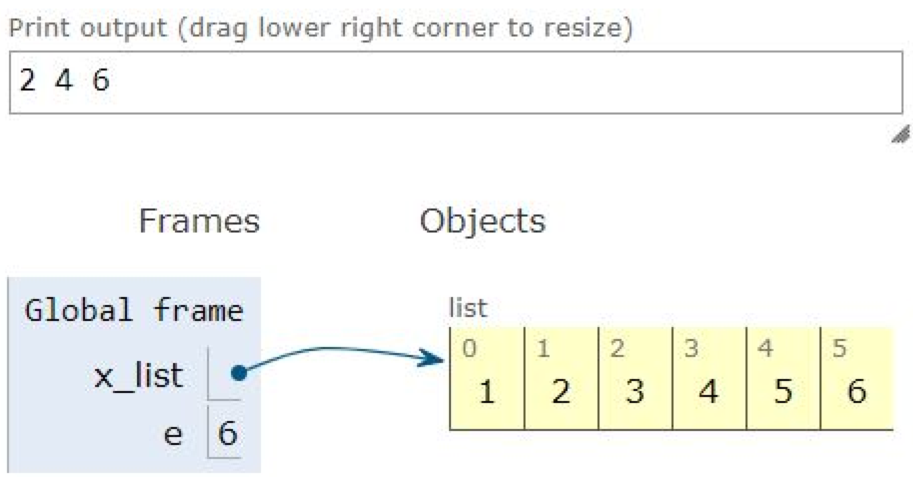
```
x           = 5 /  2
x_floor     = 5 // 2
x_remainder = 5 % 2

x_tuple     = divmod(5, 2)
z_tuple     = divmod(5.0, 2.1)
```



Example

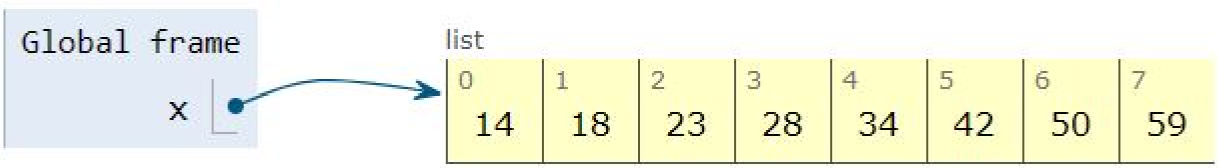
- print even numbers in a list

```
x_list = [1, 2, 3, 4, 5, 6]

for e in x_list:
    if e % 2 == 0:
        print(e, end = ' ')
```



Exercise(s):

(a) print odd numbers from a list:



(b) print list numbers divisible by 3

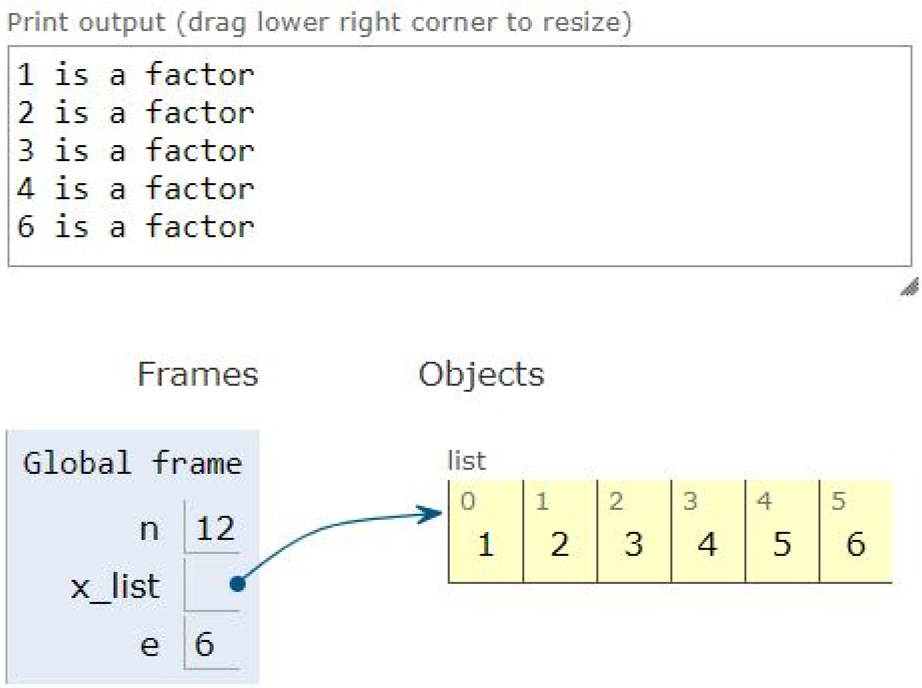
(c) print list numbers in range

$$20 \leq n \leq 50$$

Example

- print factors of 12 from a list

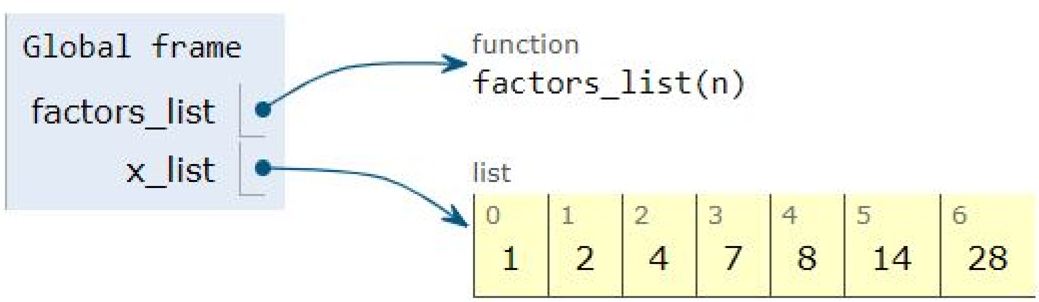
```
n = 12;
x_list = [1, 2, 3, 4, 5, 6]

for e in x_list:
    if n % e  == 0:
        print(e, 'is a factor')
```



Example: Proper Factors

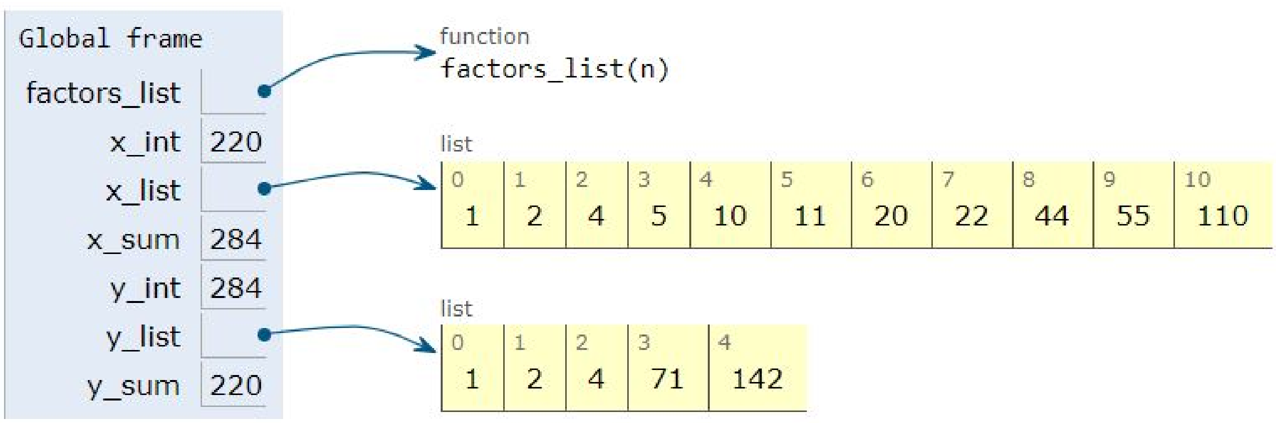
- all factors less than n

```
def factors_list(n):
    result = []
    i = 1
    while i < 1 + int(n/2):
        if n % i == 0:
            result.append(i)
        i = i + 1
    return result

x_list = factors_list(56)
```



"Amicable" Numbers

```
x_int = 220
x_list = factors_list(x_int)
x_sum = sum(x_list)

y_int = 284
y_list = factors_list(y_int)
y_sum = sum(y_list)
```



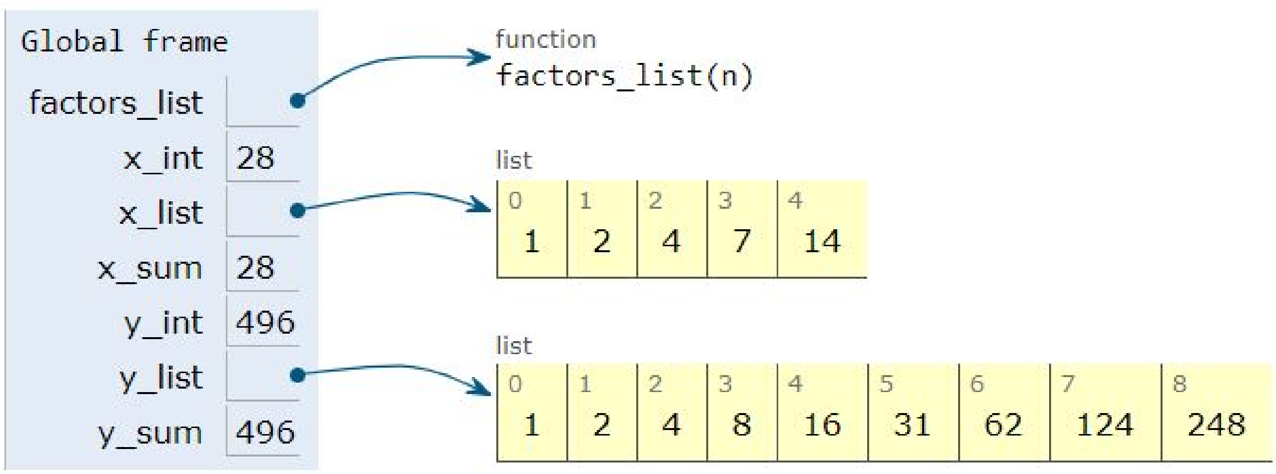
$$\text{sum of factors of } N = M$$

and

$$\text{sum of factors of } M = N$$

"Perfect" Numbers

```
x_int = 28
x_list = factors_list(x_int)
x_sum = sum(x_list)

y_int = 496
y_list = factors_list(y_int)
y_sum = sum(y_list)
```



$$\text{sum of factors of } N = N$$

Exercise:

- for each number compute (a list of) its factors:

(a) 20
(b) 30
(c) 40
(d) 60
(e) 80
(f) 96

- which numbers have the most factors?

Assignment Operators

- ”augmented” assignments
- simplified form for common computations

```
x = 5
y = 10
x += y        # x = x + y*y
x -= y        # x = x - y
x *= y        # x = x * y
x /= y        # x = x/y
x //= y       # x = x // y
x **= y       # x = x ** y
```

- no spaces before ’=’

```
x += y          # OK
x + = y         # error !!!
```

Bitwise Operations

```
x = 6
y = 12

x_bin = bin(x)
y_bin = bin(y)

x_and_y   = x & y   # same as x and y
x_or_y    = x | y   # same as x or y
x_xor_y   =  x ^ y  # same as x or y
x_not     =  ~x
x_left_2  = x << 2
y_right_2 = y >> 2
```

- defined for integers only
- shift left (right) by n: multiply (divide) by 2^n

Bitwise Operations (cont'd)

Global frame

x	6
y	12
x_bin	"0b110"
y_bin	"0b1100"
x_and_y	4
x_or_y	14
x_xor_y	10
x_not	-7
x_left_2	24
y_right_2	3

Comparison Operators

- $==, ! =, <, <=, >, >=$

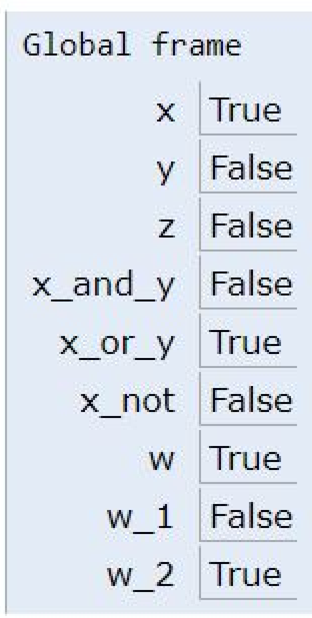
```
x, y                = 5, 100
x_equals_y          = (x == y)
x_not_equal_y       = (x != y)
x_less_than_y       = (x < y)
x_less_or_equal_y = (x <= y)
x_greater_than_y    = (x > y)
x_greater_equal_y = (x >= y)
```

Global frame	
x	5
y	10
x_equals_y	False
x_not_equal_y	True
x_less_than_y	True
x_less_or_equal_y	True
x_greater_than_y	False
x_greater_equal_y	False

Logical Comparisons

- *and*, *or*, *not*

```
x = True; y = False; z = False
x_and_y = (x and y)
x_or_y  = (x or y)
x_not   = not x

w_1 = (x or y) and z; w_2 = x or (y and z)
w = x or y and z
```



Exercise(s):

- prove the following identity:

$$\text{not}(A \text{ and } B) = (\text{not } A) \text{ or } (\text{not } B)$$

Hint: prove this for possible combinations of boolean values for A and B

Identity Comparisons

- *is*, *is not*
- true if objects are identical

```
x =  5     # object of type integer
y = 5.0    # object of type float

x_is_y      = (x is y)    # different objects
x_equals_y = (x == y)    # but same value
```

Global frame

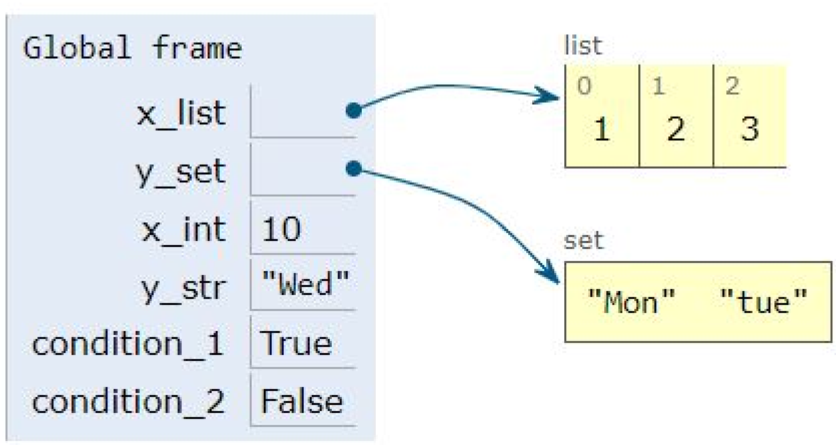
x	5
y	5.0
x_is_y	False
x_equals_y	True

- *"x is True"* or *"x == True"*?
- Python style: *"x is True"*

Membership Operators

- *in*, *not in*
- same for all collections

```
x_list  = [1,2,3]
y_set   = {'Mon', "tue"}

x_int   = 10
y_str   = 'Wed'

condition_1 = (x_int not in x_list)
condition_2 = (y_str in y_set)
```



Operators Summary

- arithmetic: $+, -, *, **, /, //, **$
- assignment:

$$=, + =, - =, * =, / =, \% =, // =$$

- bitwise: $\&, |, \hat{}\ , <<, >>$
- comparison: $==, ! =, <, <=, >, >=$
- logical: *and*, *or*, *not*
- identity: *is*, *is not*
- membership: *in*, *not in*

Random Number

```
import random

x = -1 + 2 * random.random()
y = -1 + 2 * random.random()
if (x*x + y*y)  <= 1:
    print(x, y, ' inside unit circle')
else:
    print(x, y, ' outside unit circle')
```

- use *random* module
- *random*() random value in $[0, 1]$

Exercise(s):

- generate

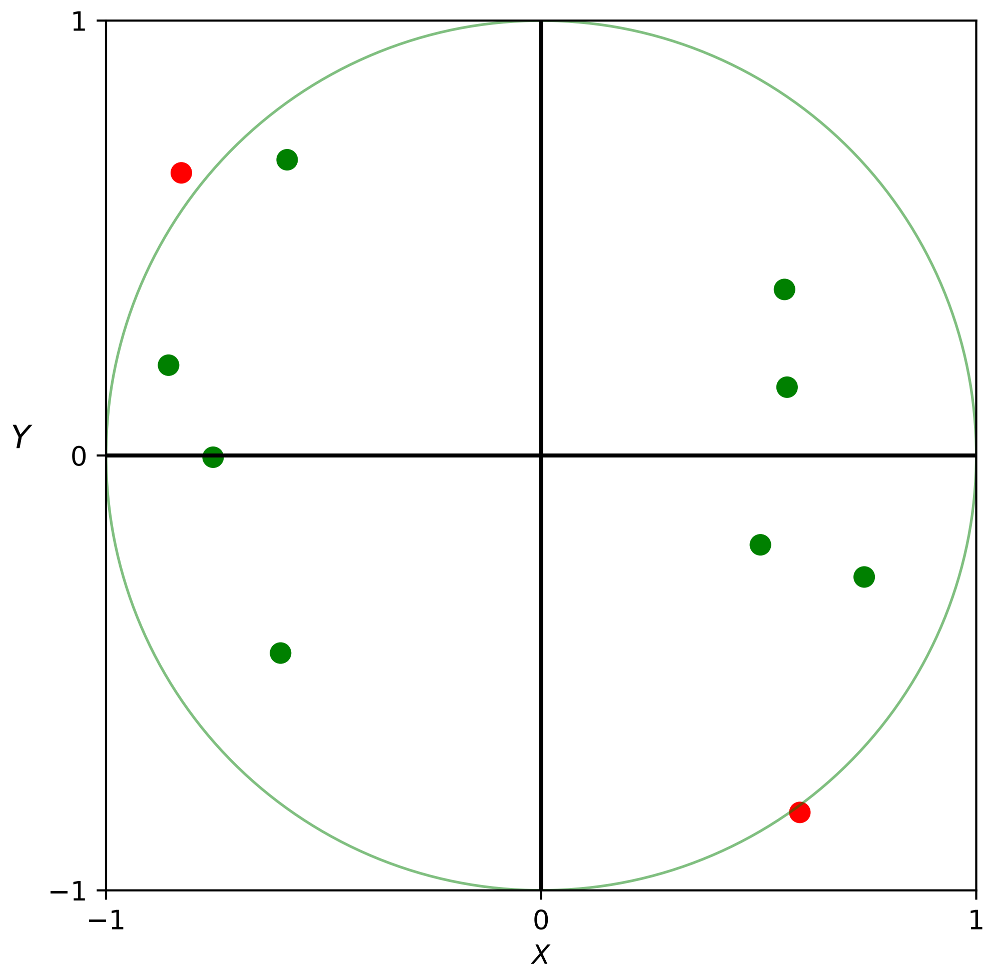
(a) 10 random numbers in range [-3, 1]

(b) 10 random integers in range [-1, 3]

(c) 10 random integers in range [-3, 1] and count negative numbers

(d) 10 random integers in range [1, 3] and count even numbers

Monte-Carlo Simulation



Simulation Details

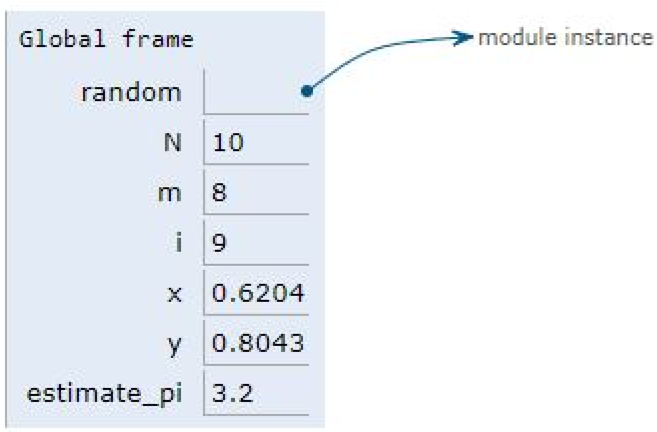
- $\pi = 3.14159265359\ldots$
- estimate π from random points
- generate N points inside square
- count m points inside circle

$$\frac{m}{N} \mapsto \frac{area(\mathrm{C}ircle)}{area(\mathrm{S}quare)} = \frac{\pi}{4}$$

- approximate $\pi \approx 4m/N$

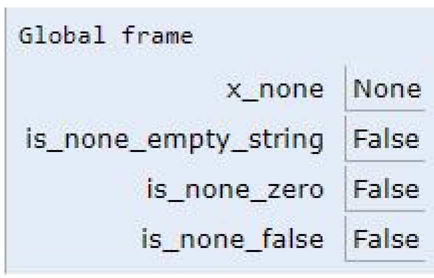
Code for Simulation

```python
# estimate pi = 3.14159 by Monte_Carlo
import random
N = 10
m = 0
for i in range(N):
    x = -1 + 2 * random.random()
    y = -1 + 2 * random.random()
    if (x*x + y*y) <= 1:
        m = m + 1

estimate_pi = 4.0 * m / N
```



None Object

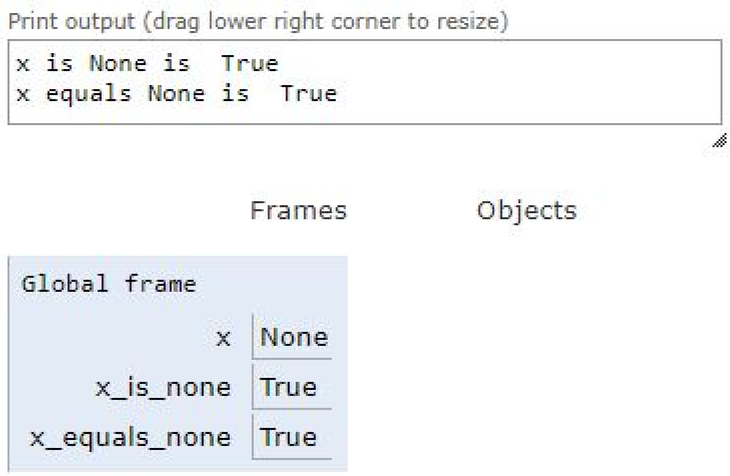
```
x_none = None
is_none_empty_string = (x_none == "")
is_none_zero         = (x_none == 0)
is_none_false        = (x_none == False)
```



- special *null* object
- type: *NoneType*
- a singleton (one instance !!!)
- can assign to any variable

None Example

```
x = None

x_is_none     = (x is None)
x_equals_none = (x == None)
print('x is None is ', x_is_none)
print('x equals None is ', x_equals_none)
```



- both comparisons are valid
- (x is None) is preferred

Summary:

- data types are basic building blocks
- objects in Python, bundled with methods
- primitive types are ”indvisible”
- collection types can contain other data types
- new objects can be created using operations
- many operations are available
- many operations have the same syntax for multiple data types