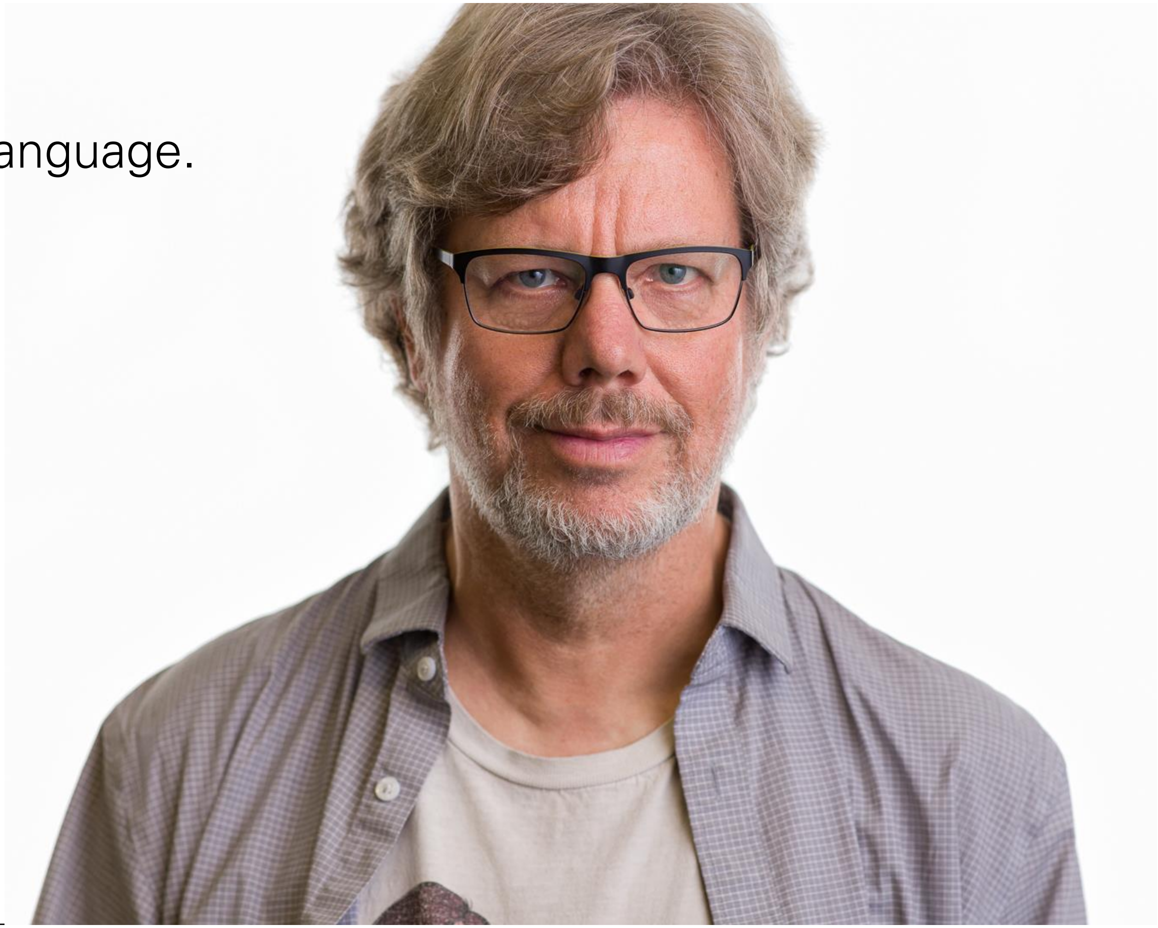
Python For Engineers I

Why Python?

- Used in every software industry.
  - From web development to even embedded devices & Raspberry Pi’s.
- Easy-to-pick-up language that only gets as complex as one requires.
- Writing Python code feels like Pseudocode.
- Great language to prototype/test out software concepts and ideas.
- *Nicer to use than Java and MATLAB*.
- Complete Python environment can be embedded into a C(++) program.
  - Allows for faster development time.
  - Wider audience reach to make plugins for an application.

Little bit of History

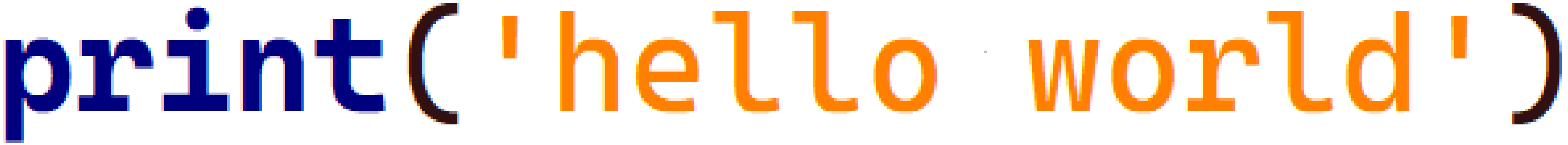
- Brainchild of **Guido Van Rossum** in the Netherlands.
- Conceived in the **late 1980s.**
- Created as a successor of the ABC Programming Language.
- **Python version 1.0 came out January 26, 1994**
  - Python is technically older than Java!
  - **Java version 1.0 was released January 23, 1996**
- Boomed in popularity in 2003+
- *Core Philosophy*:
  - **Beautiful is better than ugly.**
  - **Explicit is better than implicit.**
  - **Simple is better than complex.**
  - **Complex is better than complicated.**
  - **Readability counts.**



Starting a Python program

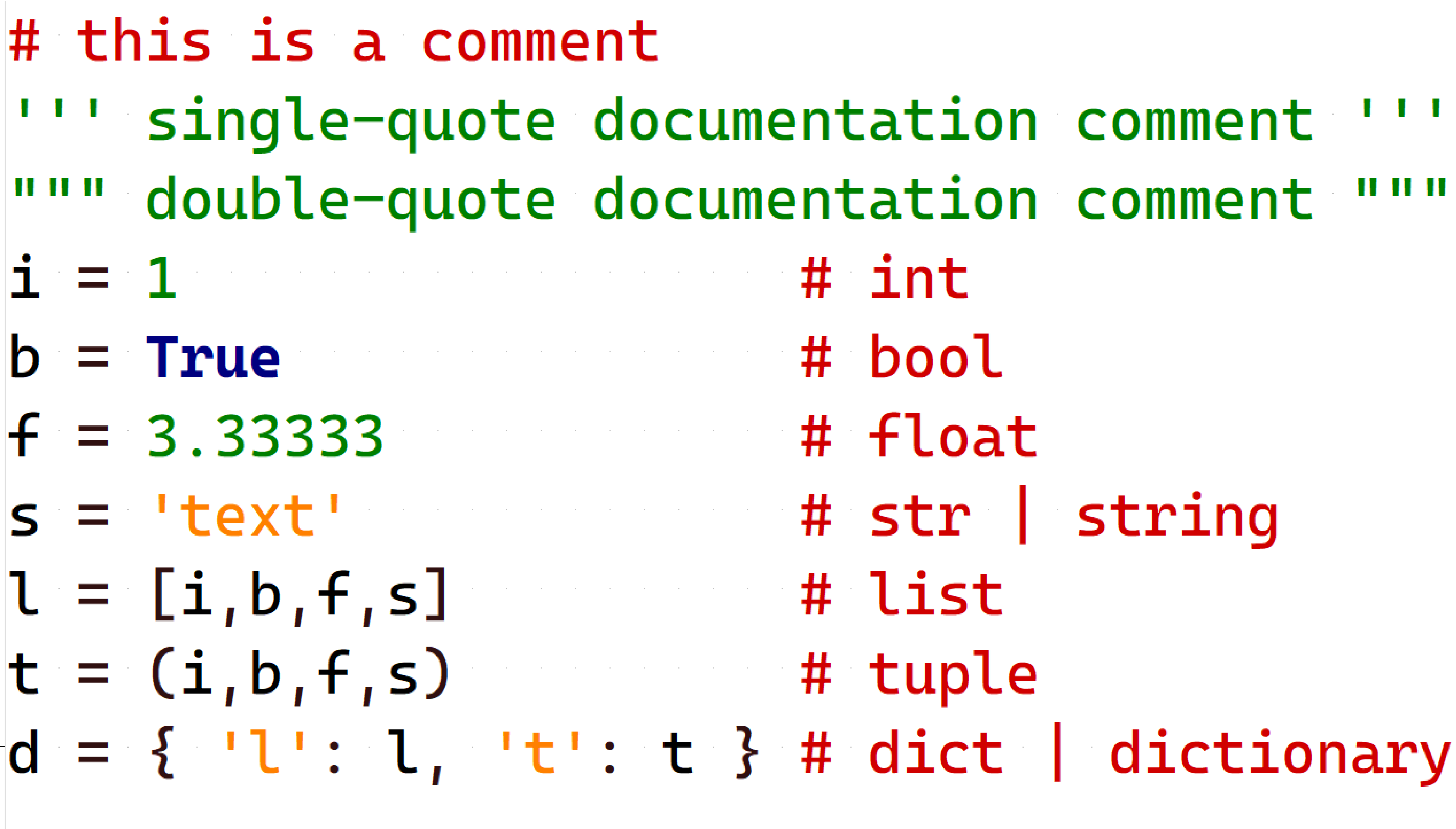
- Starting Python programs is extremely simple.
- If you prefer Online IDEs (integrated development environments, check out replit.com or onlinegdb.com
- If you want an Offline IDE, you will need to install Python for your system, to do this check out python.org/downloads/
  - After installing Python, you then need an offline IDE: VSCode, PyCharm, Geany, Spyder, etc.
- Python programs are segmented into individual files, containing Python code, called ***scripts***.
- Python scripts always (and should) end with a **.py** file extension.
- Python programs work from a starter script which either contains all the code required *or* kicks off code that's organized in other Python scripts.
- All code in Python is sequential* and flows from top to bottom as the order of execution.

Our first Python program



```python
print('hello world')
```

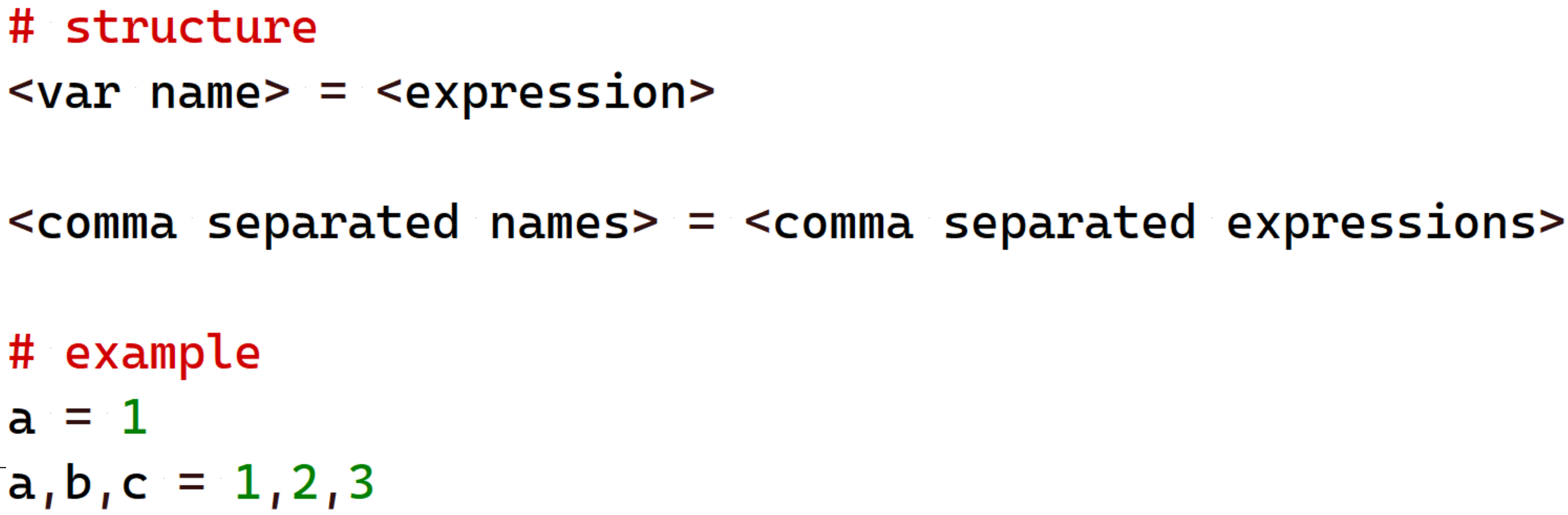
Basic Data Types of Python



```
# this is a comment
''' single-quote documentation comment '''
""" double-quote documentation comment """
i = 1                  # int
b = True               # bool
f = 3.33333            # float
s = 'text'             # str | string
l = [i,b,f,s]          # list
t = (i,b,f,s)          # tuple
d = { 'l': l, 't': t } # dict | dictionary
```

Creating Variables

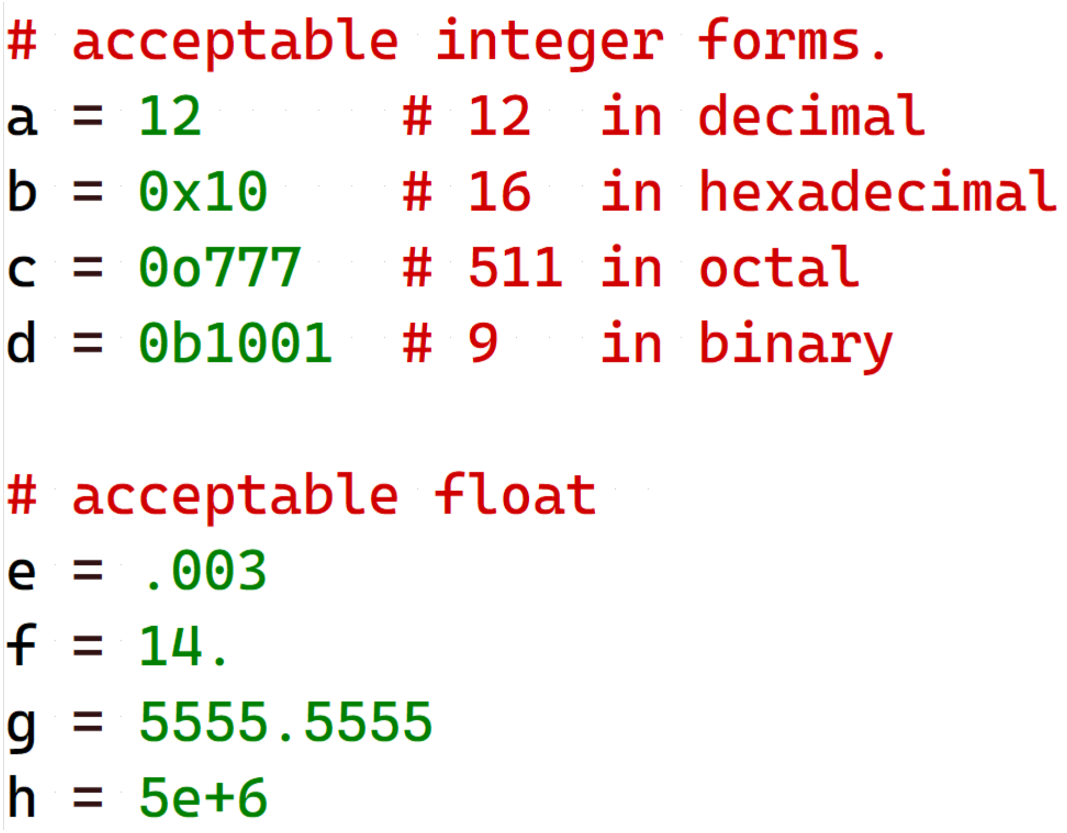
- Creating variables is as easy as using a name and then giving it a value directly.
- Everything in Python is an object, including variables!
- Objects are chunks of memory that store our data.
    - typically have a set of actions/behaviors associated with the data.



```
# structure
<var name> = <expression>

<comma separated names> = <comma separated expressions>

# example
a = 1
a,b,c = 1,2,3
```

Types of Numbers

- Integers:
  - integer numbers, positive and negative.
  - Usable in decimal, hex, binary, and octal numeral form
  - integers has no maximum limit!
    - the larger your integer values the more memory required to represent the value.
- Floats:
  - Represents rational, decimal-point numbers.
  - Has a limit of 64-bits.
- Complex:
  - Same as floats but numbers require a '**`j`**' suffix like: `**`1j`**`.
  - Real and Imaginary parts can be retrieved:
    - **`1j.real or 1j.imag`**



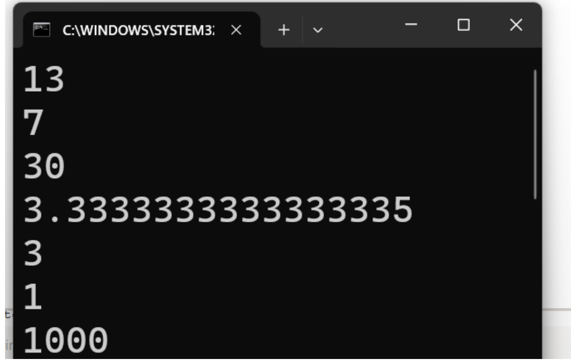
```
# acceptable integer forms.
a = 12      # 12  in decimal
b = 0x10    # 16  in hexadecimal
c = 0o777   # 511 in octal
d = 0b1001  # 9   in binary

# acceptable float
e = .003
f = 14.
g = 5555.5555
h = 5e+6
```

Intro to Expressions

- Expressions are lines of code in which the computation of *can* produce a value.
  - In short, any line of code that interacts with numbers/data and/or variables/objects.
- Types of Expressions in Python:
  - Function calls
  - Arithmetic operations
  - Logical operations
  - Relational/Comparison operations
  - Accessing data through objects.

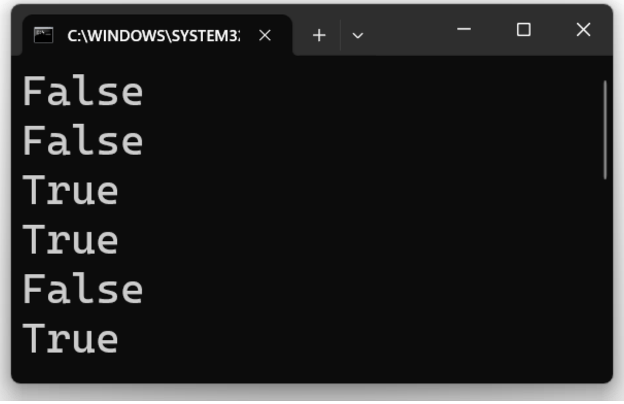
Expressions – Arithmetic & Relational

```
print(10 + 3)
print(10 - 3)
print(10 * 3)
print(10 / 3)
print(10 // 3) # '//' means integer division.
print(10 % 3)  # '%' is modulo.
print(10 ** 3) # '**' means power!
```



```
13
7
30
3.3333333333333335
3
1
1000
```

```
print(10 <  3)   # 10 less than 3
print(10 <= 3)   # 10 less than-equal to 3
print(10 >  3)   # 10 greater than 3
print(10 >= 3)   # 10 greater than-equal 3
print(10 == 3)   # 10 equal to 3
print(10 != 3)   # 10 not equal to 3
```



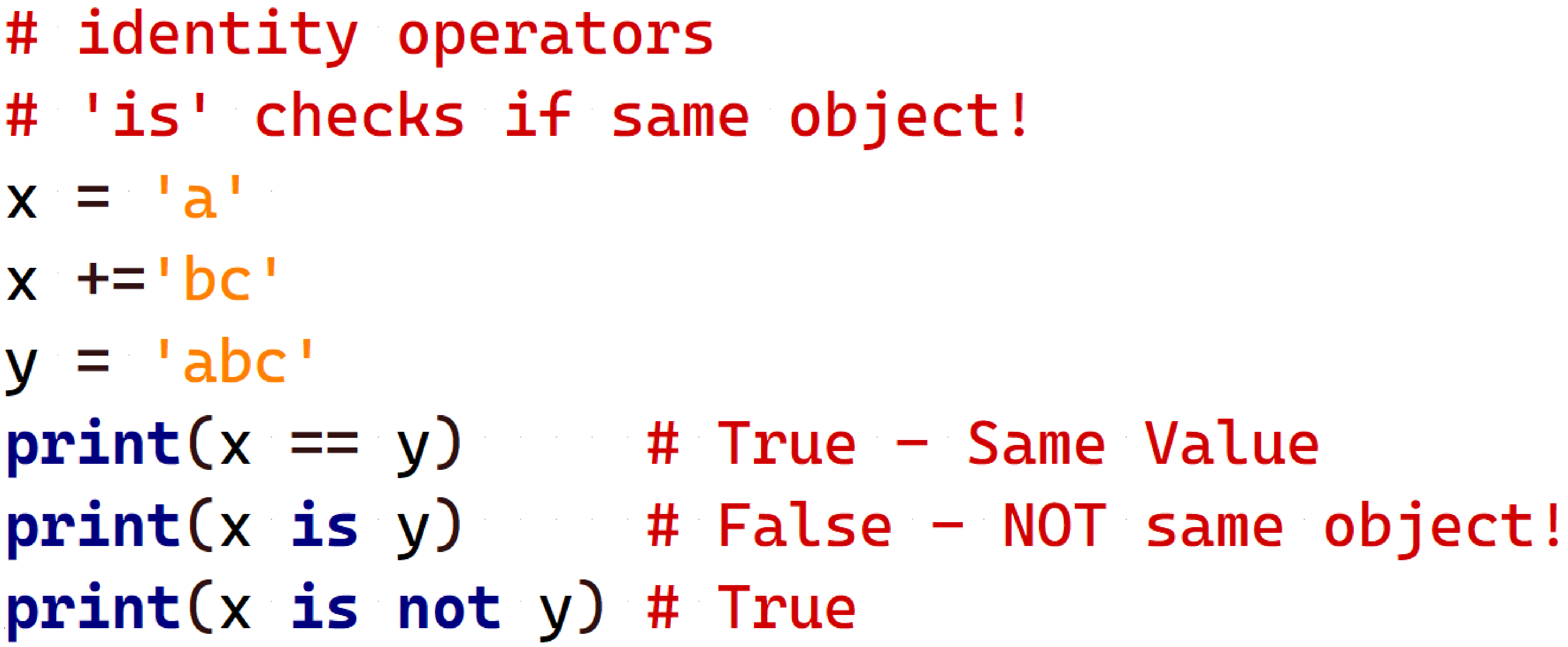
```
False
False
True
True
False
True
```

Expressions – Logical & [Bitwise] Arithmetic

```
# logical operators
a, b = True, False
print(a and b)    # False
print(a or b)     # True
print(not a)      # False
print(not b and (a or b)) # True
```

```
# bitwise operators
# only works for int objects.

# AND operator - &
print(1 & 3) # prints 1
# OR operator - |
print(1 | 4) # prints 5
# XOR operator - ^
print(1 ^ 3) # prints 2
# NOT operator - ~
print(~1)    # prints -2
```

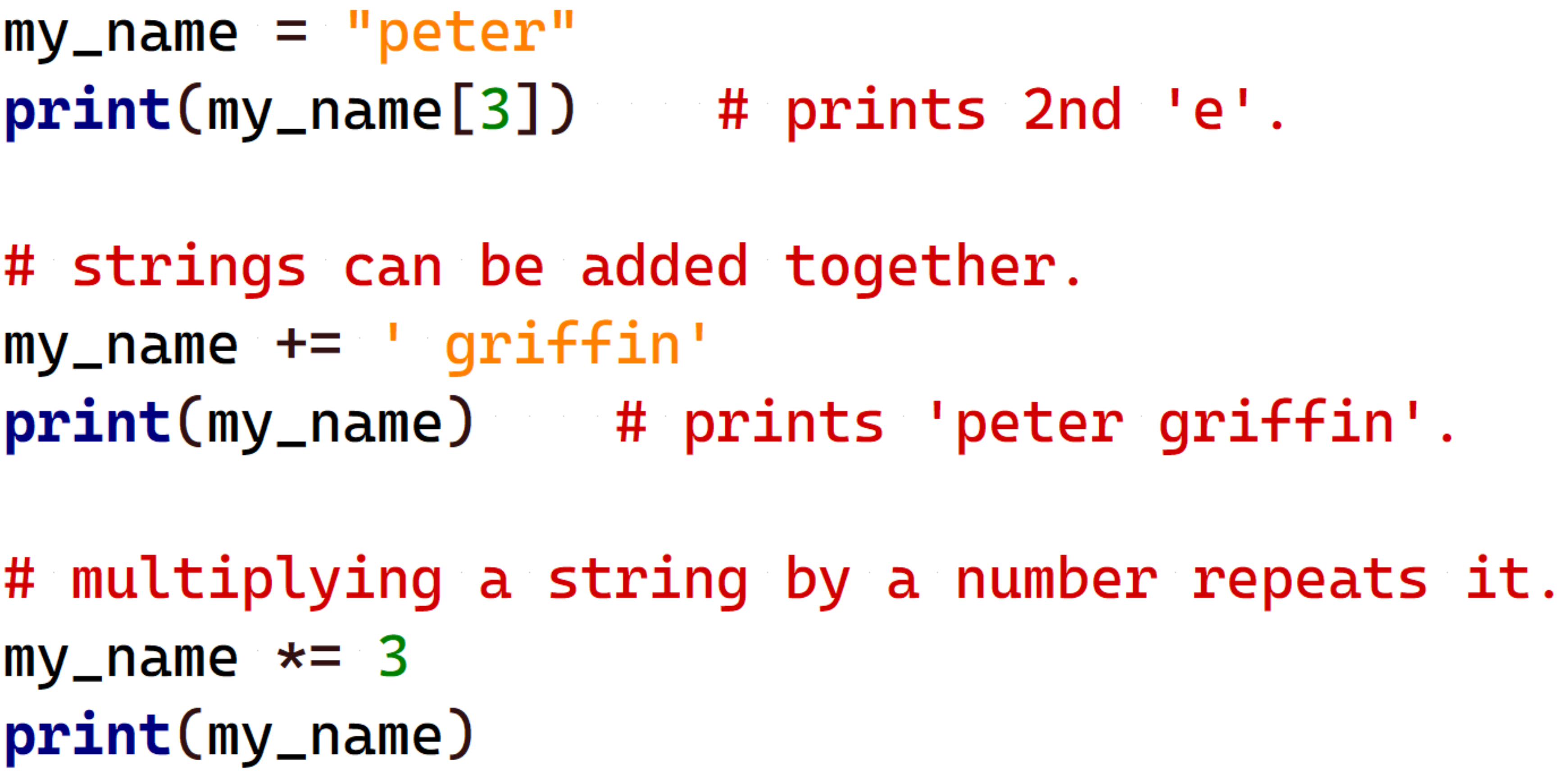
Expressions - Identity Operators



```
# identity operators
# 'is' checks if same object!
x = 'a'
x +='bc'
y = 'abc'
print(x == y)     # True - Same Value
print(x is y)     # False - NOT same object!
print(x is not y) # True
```

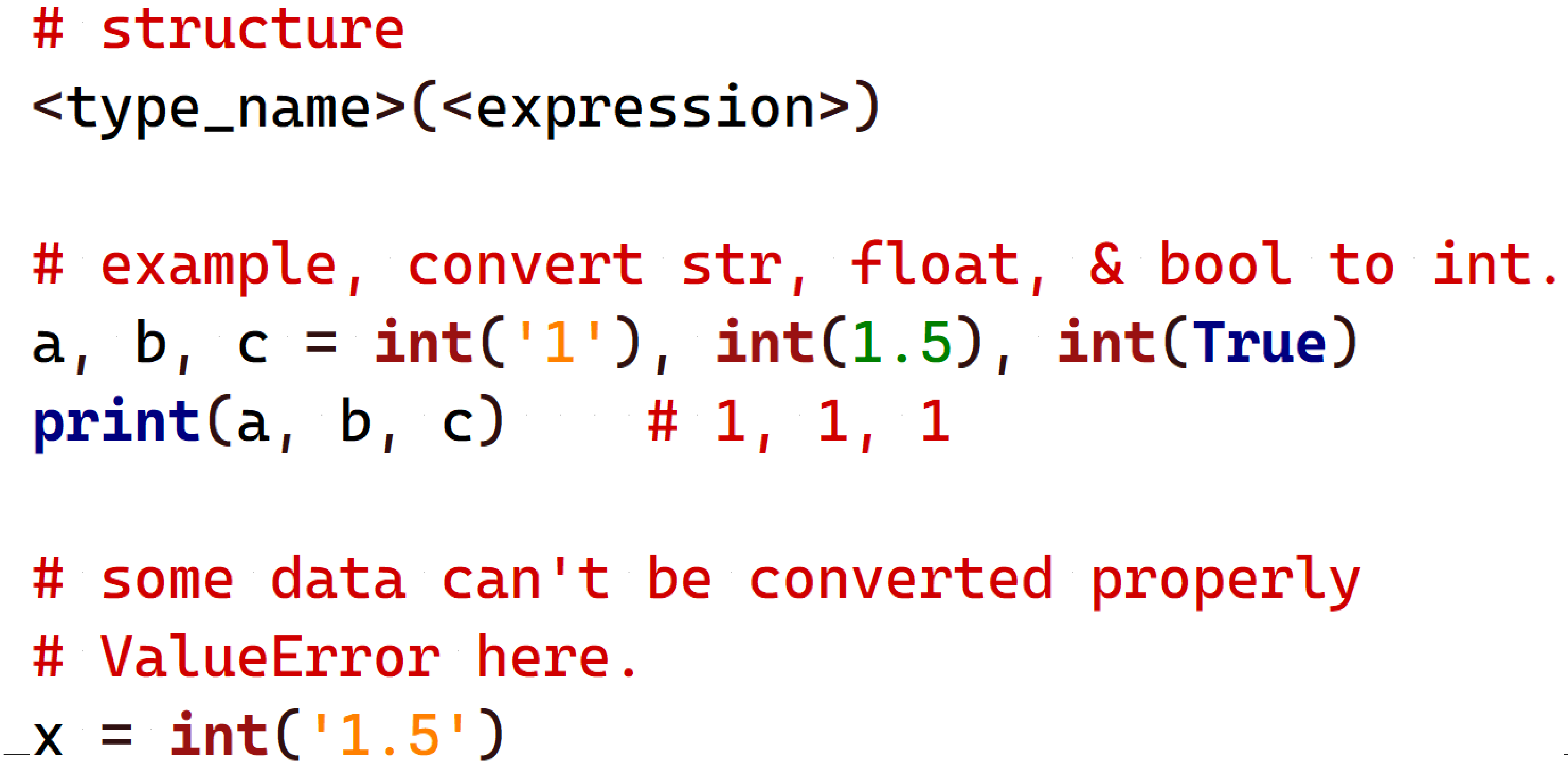
Strings - Basics

- Can be created using single or double quotes!
  - This allows using the other quote when using a specific one to indicate the string.
- Strings can also be indexed like an array.



```
my_name = "peter"
print(my_name[3])    # prints 2nd 'e'.

# strings can be added together.
my_name += ' griffin'
print(my_name)    # prints 'peter griffin'.

# multiplying a string by a number repeats it.
my_name *= 3
print(my_name)
```

Expressions - Data Conversion

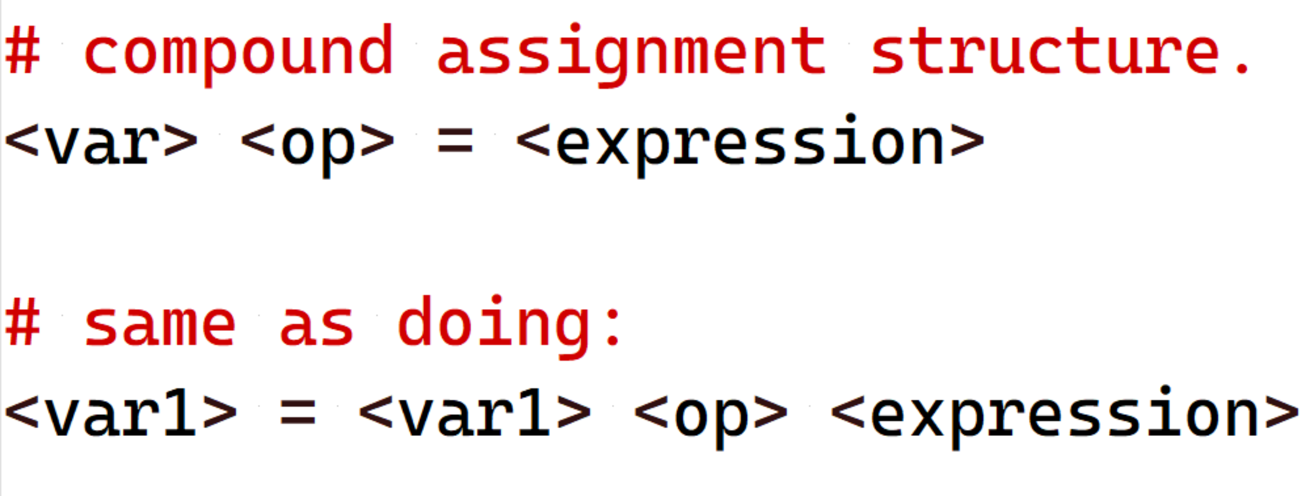


```
# structure
<type_name>(<expression>)

# example, convert str, float, & bool to int.
a, b, c = int('1'), int(1.5), int(True)
print(a, b, c)     # 1, 1, 1

# some data can't be converted properly
# ValueError here.
x = int('1.5')
```

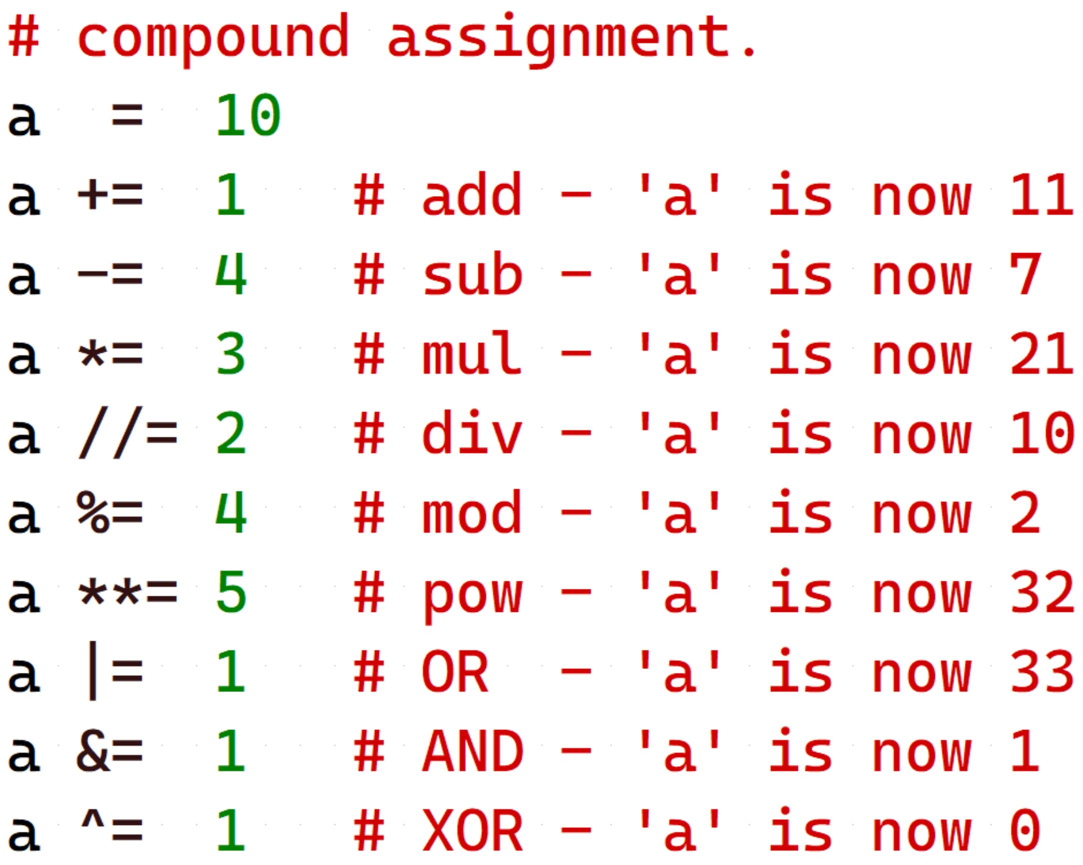
Intro to Statements

- Statements are lines of code that conveys an action to be carried out.
- Statements can contain other statements.
  - These are called *compound statements*.
- Statements can also contain expressions.
  - A standalone expression as a statement are called *expression statements*.
  - Most expressions in code are usually expression statements involving variables.
  - Creating variables is an example of an expression statement.
    - However, they’re not proper expressions!

Statements - Compound Assignment

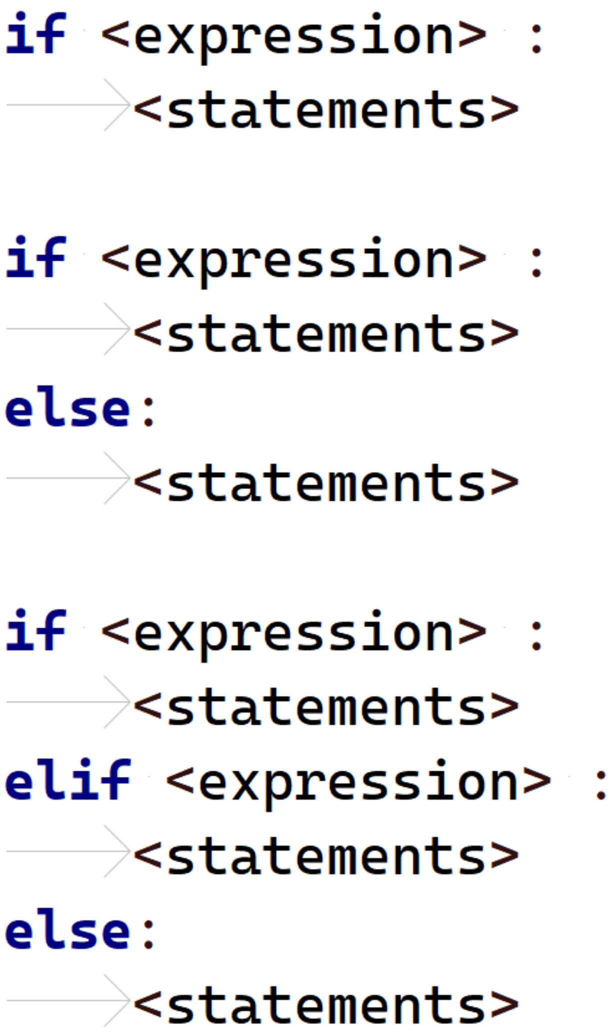


```
# compound assignment structure.
<var> <op> = <expression>

# same as doing:
<var1> = <var1> <op> <expression>
```

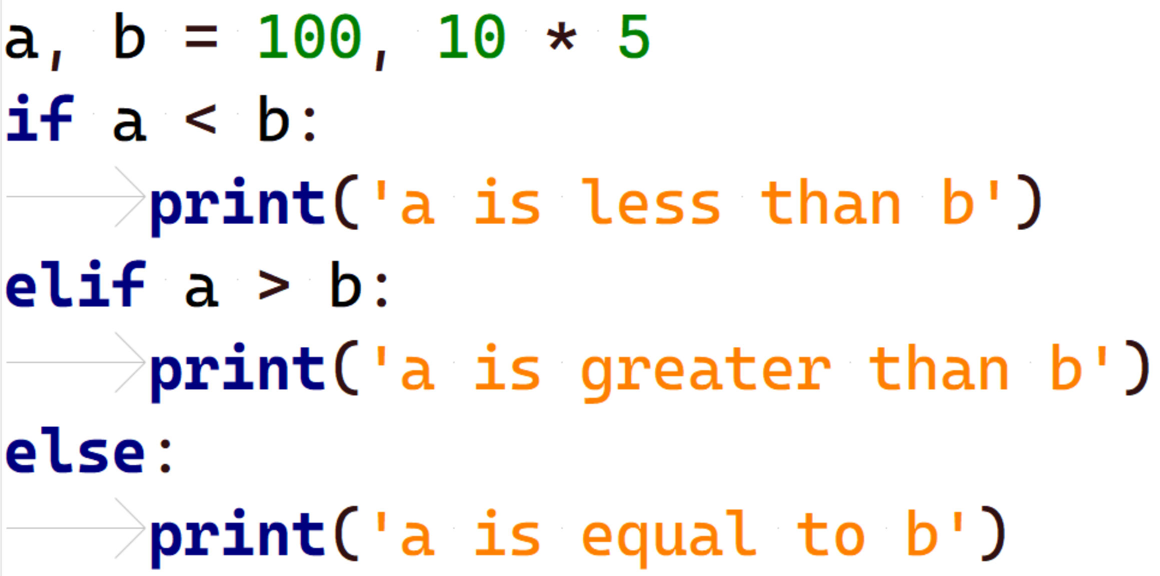


```
# compound assignment.
a  =  10
a +=  1    # add - 'a' is now 11
a -=  4    # sub - 'a' is now 7
a *=  3    # mul - 'a' is now 21
a //= 2    # div - 'a' is now 10
a %=  4    # mod - 'a' is now 2
a **= 5    # pow - 'a' is now 32
a |=  1    # OR  - 'a' is now 33
a &=  1    # AND - 'a' is now 1
a ^=  1    # XOR - 'a' is now 0
```

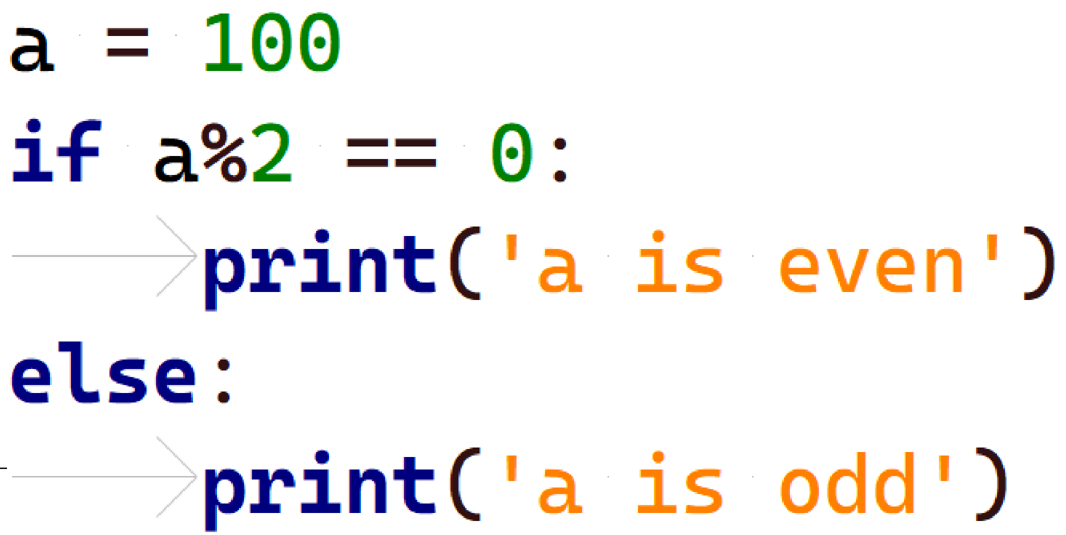
Control Flow – If Statements



```
if <expression> :
    <statements>

if <expression> :
    <statements>
else:
    <statements>

if <expression> :
    <statements>
elif <expression> :
    <statements>
else:
    <statements>
```

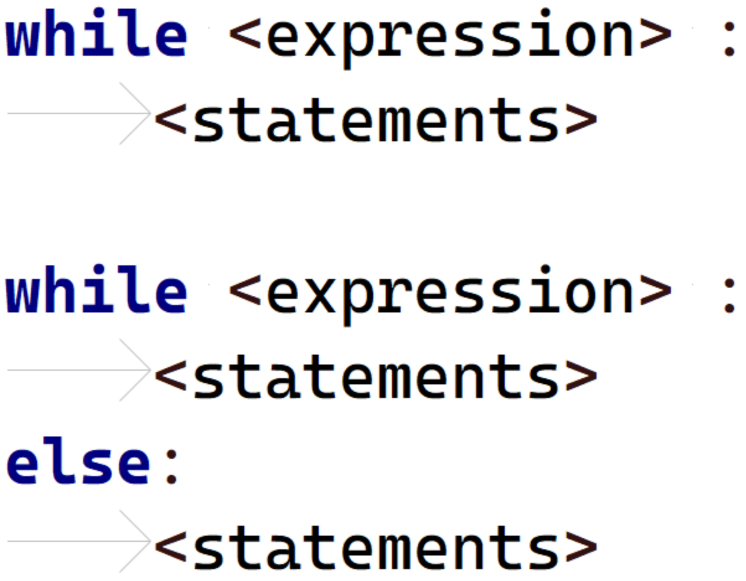


```python
a, b = 100, 10 * 5
if a < b:
    print('a is less than b')
elif a > b:
    print('a is greater than b')
else:
    print('a is equal to b')
```

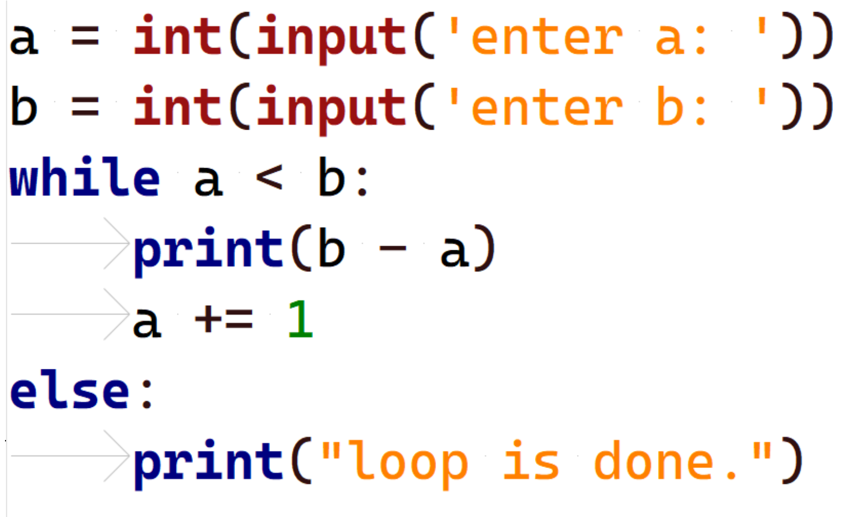


```python
a = 100
if a%2 == 0:
    print('a is even')
else:
    print('a is odd')
```

Control Flow – While Loop Statements

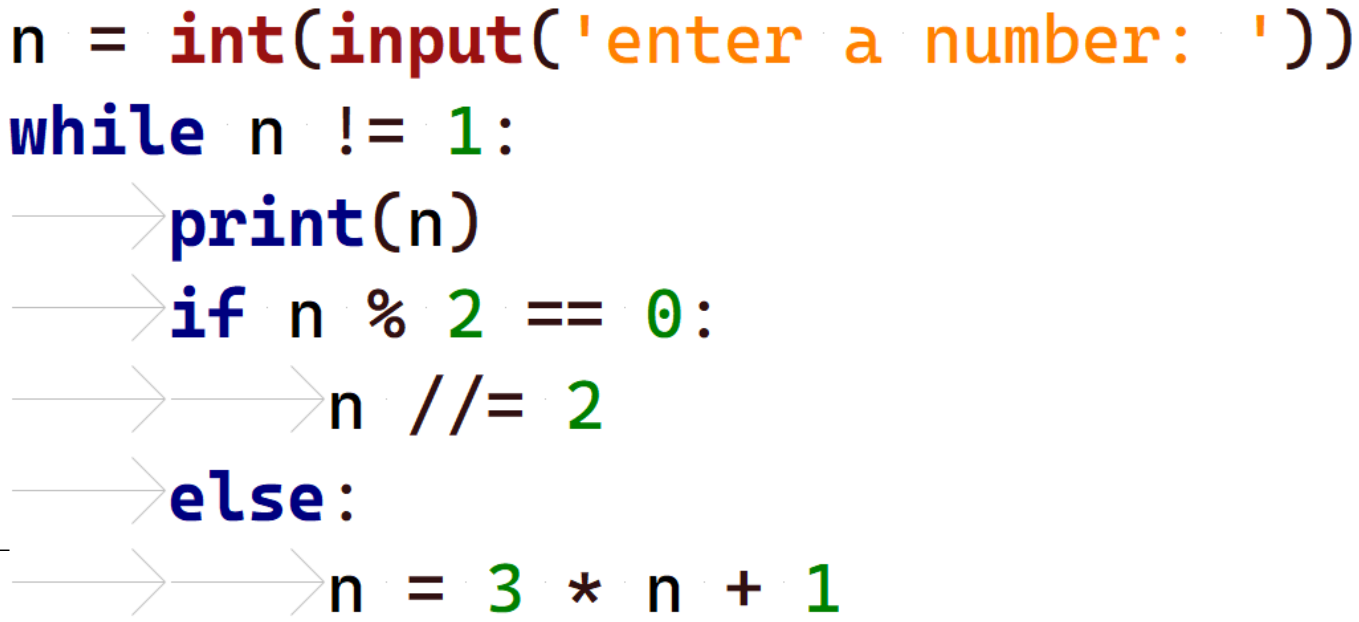


```
while <expression> :
    <statements>

while <expression> :
    <statements>
else:
    <statements>
```



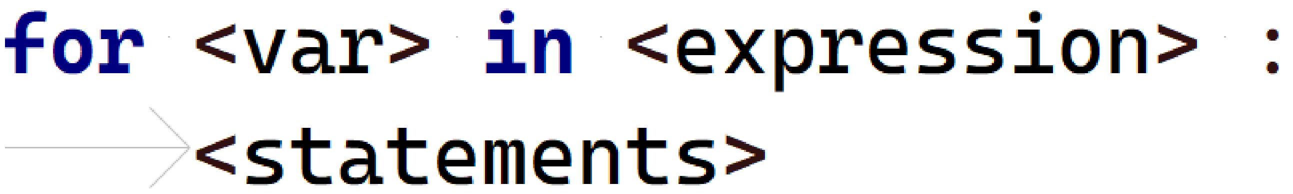
```
a = int(input('enter a: '))
b = int(input('enter b: '))
while a < b:
    print(b - a)
    a += 1
else:
    print("loop is done.")
```

- *Best to use for data that has an unknown end.*
- Example with 3n+1 is called the Collatz Conjecture.
- The ‘else’ part runs after the loop finishes.

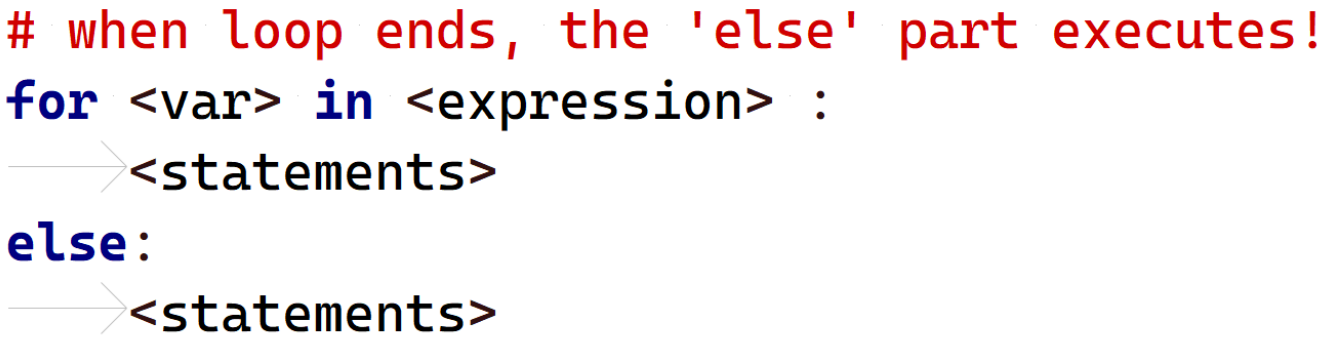


```
n = int(input('enter a number: '))
while n != 1:
    print(n)
    if n % 2 == 0:
        n //= 2
    else:
        n = 3 * n + 1
```

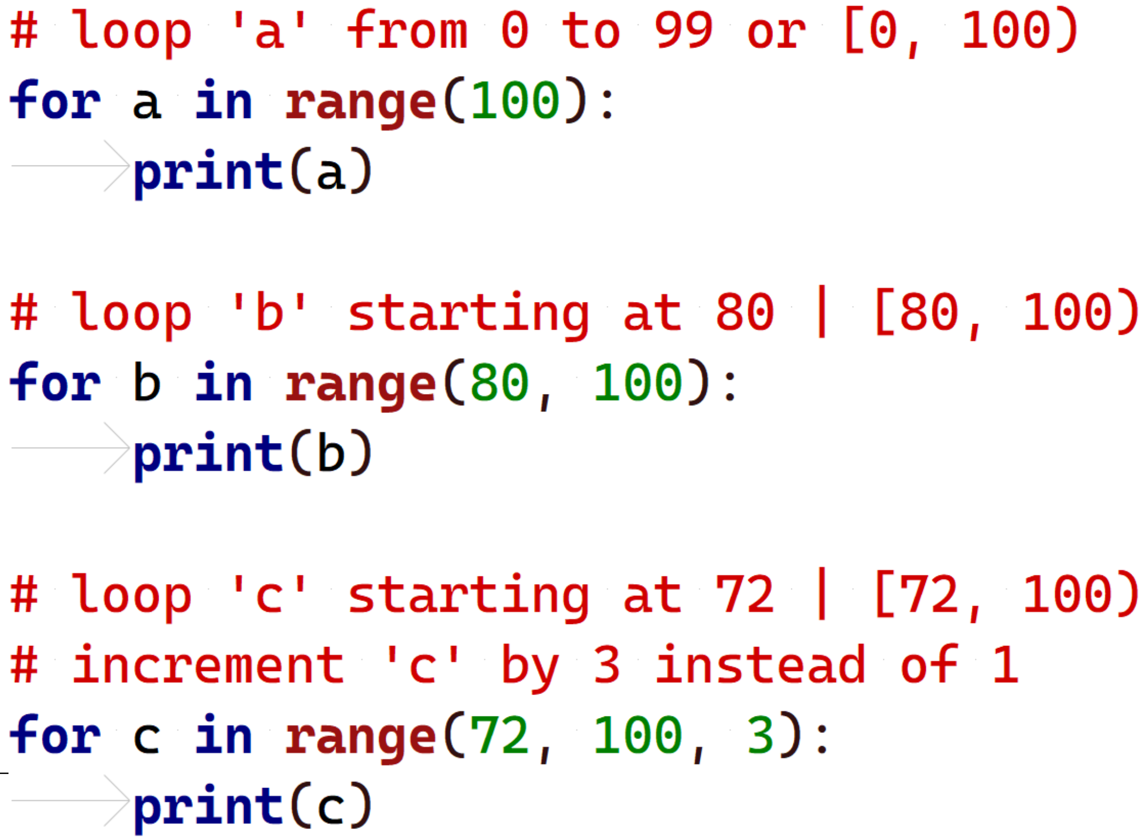
Control Flow – For Loop Statements



```
for <var> in <expression> :
    <statements>
```

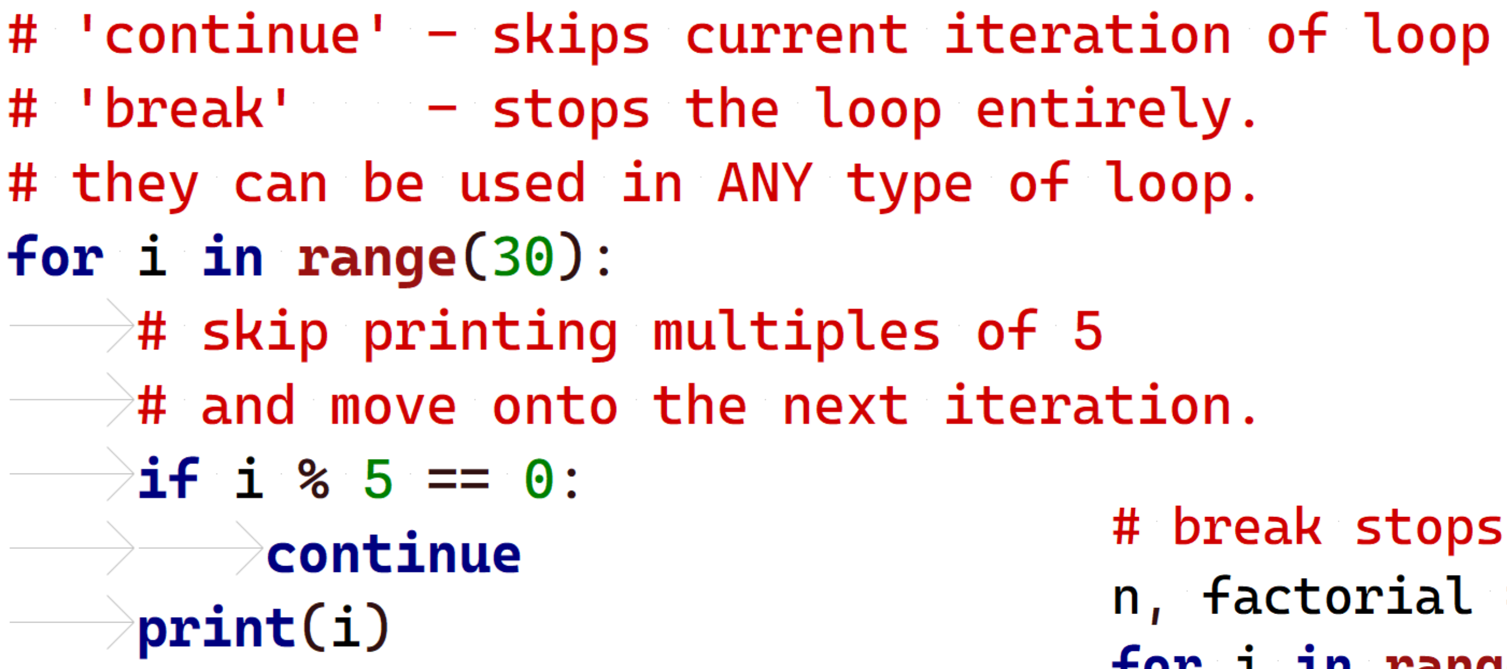


```
# when loop ends, the 'else' part executes!
for <var> in <expression> :
    <statements>
else:
    <statements>
```



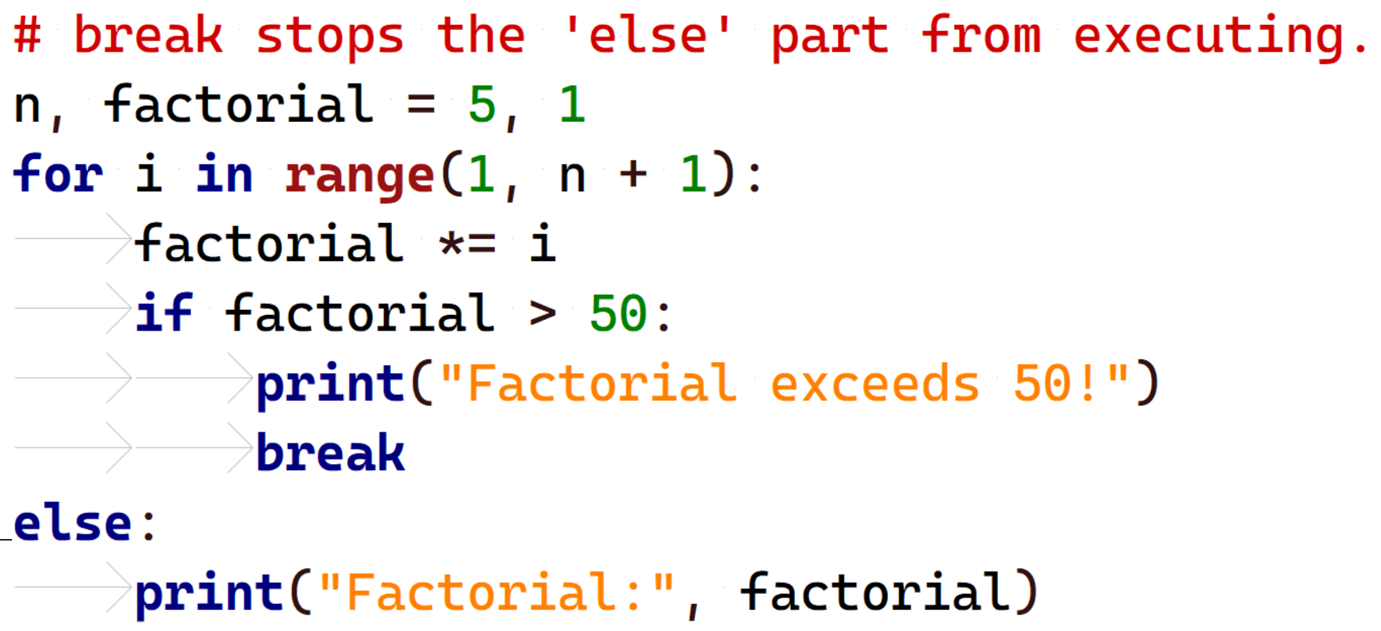
```
# loop 'a' from 0 to 99 or [0, 100)
for a in range(100):
    print(a)

# loop 'b' starting at 80 | [80, 100)
for b in range(80, 100):
    print(b)

# loop 'c' starting at 72 | [72, 100)
# increment 'c' by 3 instead of 1
for c in range(72, 100, 3):
    print(c)
```

Control Flow – Loop Control



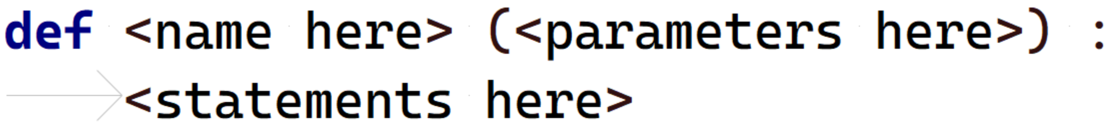
```python
# 'continue' - skips current iteration of loop
# 'break'    - stops the loop entirely.
# they can be used in ANY type of loop.
for i in range(30):
    # skip printing multiples of 5
    # and move onto the next iteration.
    if i % 5 == 0:
        continue
    print(i)
```

- **break** and **continue**.
- Can ONLY be used in a loop, any kind of loop.

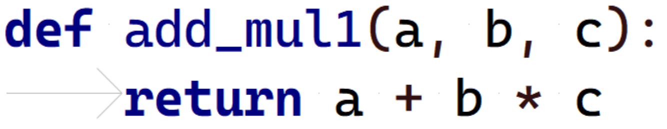


```python
# break stops the 'else' part from executing.
n, factorial = 5, 1
for i in range(1, n + 1):
    factorial *= i
    if factorial > 50:
        print("Factorial exceeds 50!")
        break
else:
    print("Factorial:", factorial)
```

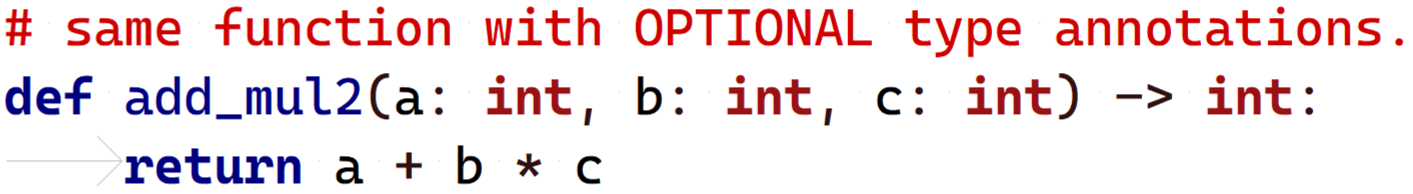
Functions I



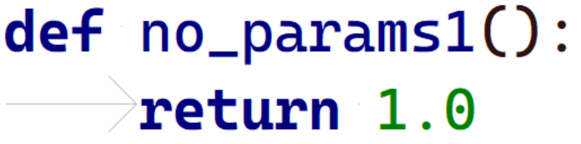
```
def <name here> (<parameters here>) :
    <statements here>
```



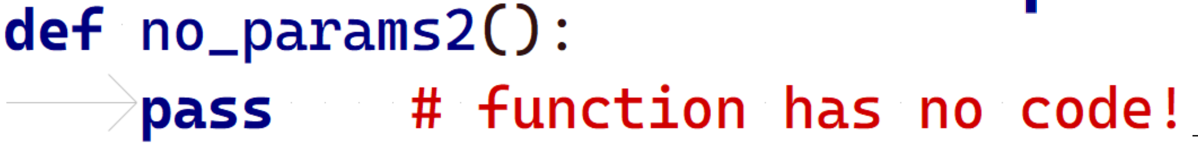
```
def add_mul1(a, b, c):
    return a + b * c
```



```
# same function with OPTIONAL type annotations.
def add_mul2(a: int, b: int, c: int) -> int:
    return a + b * c
```

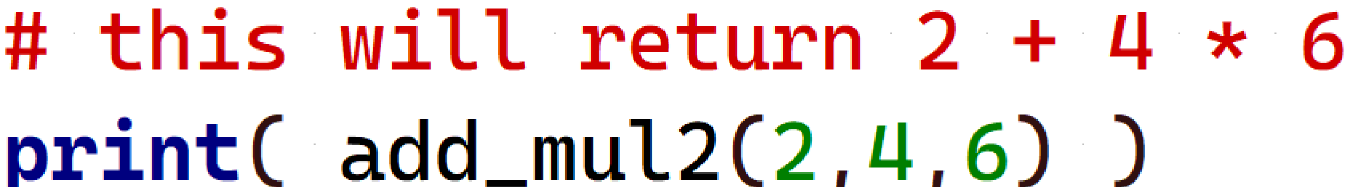


```
def no_params1():
    return 1.0
```



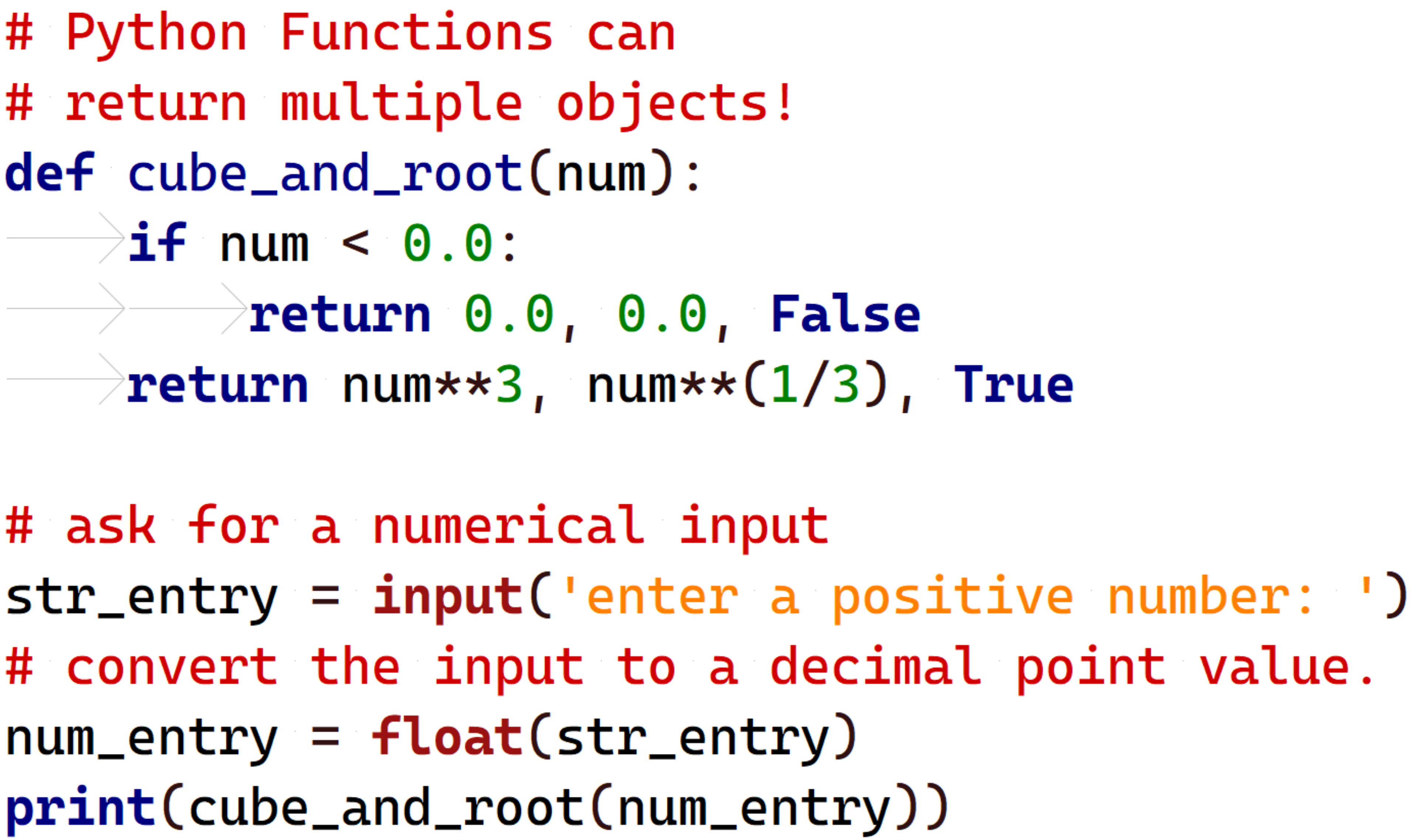
```
def no_params2():
    pass    # function has no code!
```

- Named segments of code.
- Allows us to wrap commonly used code into a reuseable package.
- Remember that Function calls are expressions!



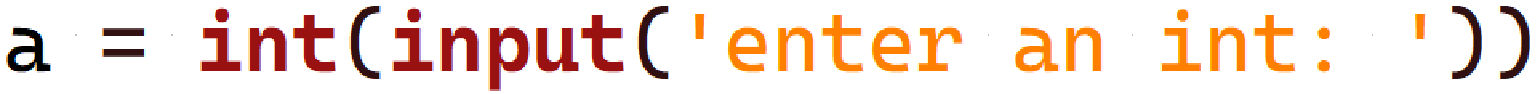
```
# this will return 2 + 4 * 6
print( add_mul2(2,4,6) )
```

Functions II



```python
# Python Functions can
# return multiple objects!
def cube_and_root(num):
    if num < 0.0:
        return 0.0, 0.0, False
    return num**3, num**(1/3), True

# ask for a numerical input
str_entry = input('enter a positive number: ')
# convert the input to a decimal point value.
num_entry = float(str_entry)
print(cube_and_root(num_entry))
```

Word Problems Set A



```
a = int(input('enter an int: '))
```

1. Square the value of **a** then print **a**.
2. if **a** is over 50, add 5 more to it then print **a**.
3. If **a** is negative, multiply **a** with -1.
4. While **a** is less than 10, print **a** then increase **a** by 2.

Word Problems Set B

```
a = int(input('enter an int #1: '))
b = int(input('enter an int #2: '))
```

1. Print the sum of **a** and **b**.
2. Print the difference of **a** and **b**.
3. Print if **a** is greater than **b**.
4. If **a**, multiplied by 2, is greater than **b**, print that **a** is bigger. Else, print **b** is greater.
5. Swap the values of **a** and **b**.
6. Print the multiplication of **a** and **b** if both **a** and **b** are even, print the division of **a** and **b** if they’re both odd, print the addition of **a** and **b** if **a** is even but **b** is odd, print the subtraction of **a** and **b** if **a** is odd but **b** is even.

Abstract Word Problem

- Using `input()`, try to create a small program that doesn't stop running (basic interactive program) unless a user specifically wants to quit.
  - Don't forget you can give `input` a prompt message as a function argument!
  - You don't necessarily need a variable but if it helps, use one.
  - Don't focus on efficiency or think there's only one way to accomplish this.

End of
Python For Engineers I

- Thank you for attending.
- Next Time: Python for Engineers II.
    - Lists, Tuples, & Dictionaries!
    - Object Oriented Python features.
    - Advanced string operations.
    - Lambda Functions.
    - Using libraries.