# Variables

## #2


Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

# Programming

Computers can be annoyingly literal. If you don't tell them exactly what you want them to do, they are likely to do the wrong thing. Try writing an algorithm for driving between two destinations. Write it the way you would for a person, and then imagine what would happen if that person were as stupid as a computer, and executed the algorithm exactly as written. (For an amusing illustration of this, take a look at the video

Exact Instructions Challenge PB&J Classroom Friendly

- https://www.youtube.com/watch?v=FN2RM-CHkuI&t=24s

# Variables

- **variable: A named piece of memory that can store a value.**
  - **Usage:**
    - **Compute an expression's result,**
    - **store that result into a variable,**
    - **and use that variable later in the program.**


- **assignment statement: Stores a value into a variable.**
  - **Syntax:**

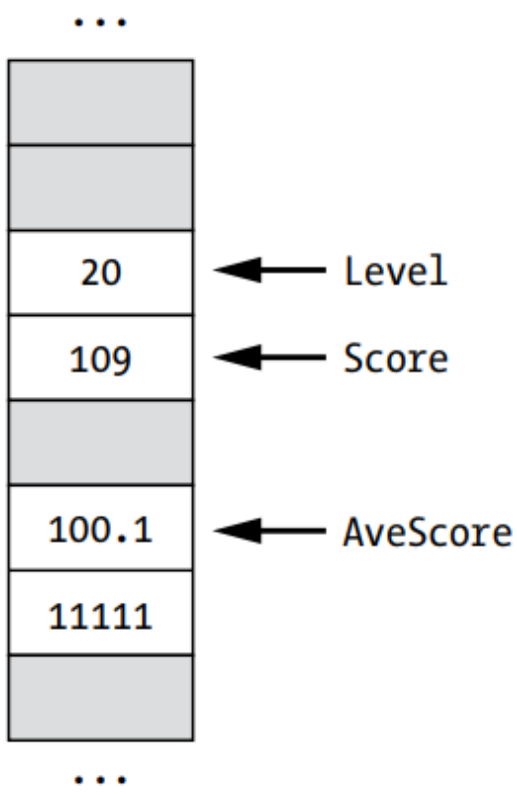
    **variable = expression**
  - **Examples:**

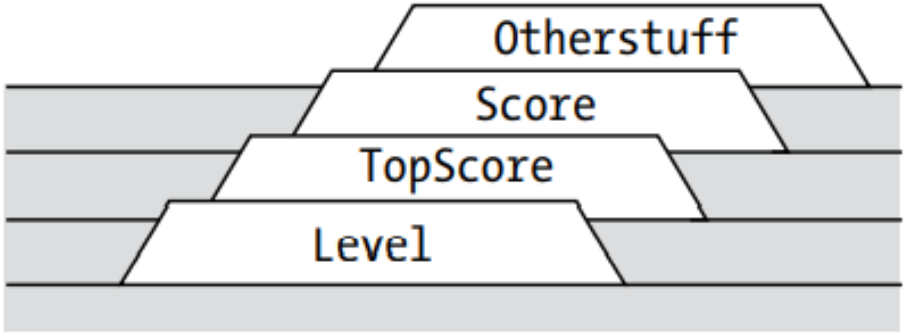
    **x = 5**

    **gpa = 3.14**

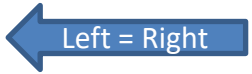
**x** → **5**  **gpa** → 3.14

# Variables


...
20
Level
109
Score
100.1
AveScore
11111
...


Otherstuff
Score
TopScore
Level

# Variables




**left-hand side = right-hand side**

**LHS = RHS**

```
>>> width = 10                  <enter>
>>> length = 5                  <enter>
>>>
>>> print(width)                <enter>
10
>>> print(length)               <enter>
5
>>> print('width')              <enter>
width
>>> print(width)                <enter>
10
>>> 25 = age                    <enter>
SyntaxError: can't assign to literal
```

# Variables

**Variables are chunks of data stored in the computers memory.**

**There are generally three types of data stored in variables. Variables can be in the form of integers or in a string. The second type of data, called a float, refers to the non-whole numbers like decimals.**

***integers*** **1, 2, 3, 4, .....**

***float*** **0.3, 1.1, 1.8, 2.5, 3.14, ....**

***string*** **“what is your name ? “, “your age is 18”, ...**

**How about 5j ? What is it?**

# Variables

- Integer objects
- Floating-point objects
- Complex number objects
- Decimal: fixed-precision objects
- Fraction: rational number objects
- Sets: collections with numeric operations
- Booleans: true and false
- Built-in functions and modules: round, math, random, etc.
- Expressions; unlimited integer precision; bitwise operations; hex, octal, and binary formats
- Third-party extensions: vectors, libraries, visualization, plotting, etc.

```
Integer: coffee_count = 5
Float: percentage_words_spelled_correctly = 21.0
Boolean: had_enough_coffee = False
```

# Variables

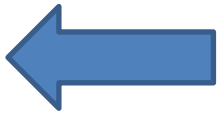
| Literal | Interpretation |
|---|---|
| 1234, −24, 0, 99999999999999 | Integers (unlimited size) |
| 1.23, 1., 3.14e-10, 4E210, 4.0e+210 | Floating-point numbers |
| 0**o**177, 0**x**9ff, 0**b**101010 | Octal, hex, and binary literals |
| 3+4j, 3.0+4.0j, 3J | Complex number literals |
| set('spam'), {1, 2, 3, 4} | Sets: construction forms |
| Decimal('1.0'), Fraction(1, 3) | Decimal and fraction extension types |
| bool(X) | True, False Boolean type and constants |

# Variables

## Numbers in Python

Python offers three different kinds of numbers with which you can work: integers , floating - point numbers (or floats ), and imaginary numbers .

```
>>> type(1)
<class 'int'>
>>> type(200000)
<class 'int'>
>>> type(99999999999999)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(5j)
<class 'complex'>
```



# Variables

**Data type Cat** **<class 'cat'>**

# Variables

## Numbers in Python

```
>>> 399 + 3020 + 1 + 3456
6876
>>> 300 - 59994 + 20
-59674
>>> 4023 - 22.46
4000.54
>>> 2000403030 * 392381727
784921595607432810
>>> 2000403030 * 3923817273929
7849215963933911604870
>>> 2e304 * 3923817273929
inf
>>> 2e34 * 3923817273929
7.847634547858e+46
```

# Variables

## Numbers in Python

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
>>> # Guess what is the answer?
>>> 0.1 + 0.1 + 0.1 - 0.3 # it must be ZERO !!
```

# Variables

## Fixed Point and Floating Point Arithmetic

| Decimal | Binary | Octal | Hexadecimal |
|---|---|---|---|
| 0 | 0000 | 0 | 0 |
| 1 | 0001 | 1 | 1 |
| 2 | 0010 | 2 | 2 |
| 3 | 0011 | 3 | 3 |
| 4 | 0100 | 4 | 4 |
| 5 | 0101 | 5 | 5 |
| 6 | 0110 | 6 | 6 |
| 7 | 0111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

Decimal numbers can be represented exactly in binary.

In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point.
End users typically would not expect 1.1 + 2.2 to display as 3.300000000000003 as it does with binary floating point.

# Variables

```
# Integer Numbers Decimal : Binary : Hexadecimal
for i in range(16):
    print(f"{f'{i:d}':>2}:{f'{i:b}':>5}:{f'{i:X}':>2}")
```

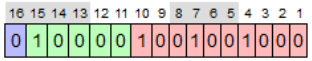
```
 0:    0: 0
 1:    1: 1
 2:   10: 2
 3:   11: 3
 4:  100: 4
 5:  101: 5
 6:  110: 6
 7:  111: 7
 8: 1000: 8
 9: 1001: 9
10: 1010: A
11: 1011: B
12: 1100: C
13: 1101: D
14: 1110: E
15: 1111: F
```

# Variables

## Numbers in Python

http://evanw.github.io/float-toy/

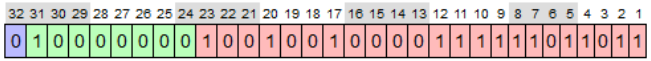
**16-bit (half)**




= 0x4248

$1 \times 2^{1} \times 1.571 = 3.141$

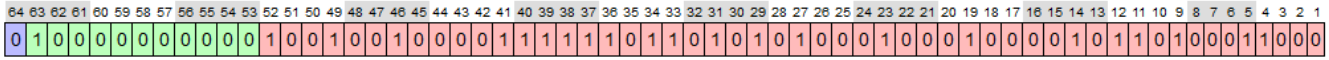
**32-bit (float)**




= 0x40490FDB

$1 \times 2^{1} \times 1.5707964 = 3.1415927$

**64-bit (double)**




= 0x400921FB54442D18

$1 \times 2^{1} \times 1.5707963267948966 = 3.141592653589793$

# Variables

## Numbers in Python

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.55111512312578e-17 # be careful !!
>>> # However, with decimals, the result can be exact:
>>> from decimal import Decimal
>>> Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3')
Decimal('0.0')
# When decimals of different precision are mixed in expressions,
# Python converts up to the largest number of decimal digits
>>> Decimal('0.1')+Decimal('0.10')+Decimal('0.10')-
Decimal('0.30')
Decimal('0.00')
```

# Variables

## Fixed Point and Floating Point Arithmetic

### Setting Decimal Precision Globally

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.14285714285714285714285714285714')
# Default: 28 digits
>>> decimal.getcontext().prec = 4
# Fixed precision
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

The decimal module provides support for fast correctly-rounded decimal floating point arithmetic. It offers several advantages over the float datatype

# Variables

## Fixed Point and Floating Point Arithmetic

```
>>> 0b0001
1
>>> 0b0010
2
>>> bin(21)
'0b10101'
>>> int('101111',2)
47
>>> 0b101111
47
>>> bin(0.5)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    bin(0.5)
TypeError: 'float' object cannot be interpreted as an integer
```

# Variables

## Mixed Types Are Converted Up

integers are simpler than floating point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first, and floating-point math yields the floating-point result

```
>>> 40 + 3.14          # Integer to float, float math/result
43.14
```

You can force the issue by calling built-in functions to convert types manually

```
>>> int(3.1415)        # Truncates float to integer
3
>>> float(3)           # Converts integer to float
3.0
```

# Variables

## Variables and Basic Expressions

Variables are created when they are first assigned values.
Variables are replaced with their values when used in expressions.
Variables must be assigned before they can be used in expressions.
Variables refer to objects and are never declared ahead of time.

```
>>> a = 3     # Name created: not declared ahead of time
>>> b = 4
>>> a + 1, a - 1 # Addition (3+1), subtraction (3-1)
(4, 2)
>>> b * 3, b / 2 # Multiplication (4*3), division (4/2)
(12, 2.0)
>>> a % 2, b ** 2    # Modulus (remainder), power (4**2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b       # Mixed-type conversions
(6.0, 16.0)
```

# Variables

## Unpacking a Sequence into Separate Variables

```
>>> p  =  (4, 5)
>>> x,  y  =  p
>>> x
4
>>> y
5
```

If there is a mismatch in the number of elements, you'll get an error. For example:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
```

# Variables

## Legal or Illegal?

**Variable names may only use letters, digits, or underscores.**

| Variable Name | Legal or Illegal? |
|---|---|
| `units_per_day` | Legal |
| `dayOfWeek` | Legal |
| `3dGraph` | Illegal. Variable names cannot begin with a digit. |
| `June1997` | Legal |
| `Mixture#3` | Illegal |

**Please give 5 Legal and 5 Illegal Variable Names**

# Python Naming Conventions

| Type of Name | Examples |
|---|---|
| Variable | `salary`, `hoursWorked`, `isAbsent` |
| Constant | `ABSOLUTE_ZERO`, `INTEREST_RATE` |
| Function or method | `printResults`, `cubeRoot`, `input` |
| Class | `BankAccount`, `SortedSet` |

Examples of Python Naming Conventions

# Python Naming Conventions

**Do NOT use these words as a variable name**

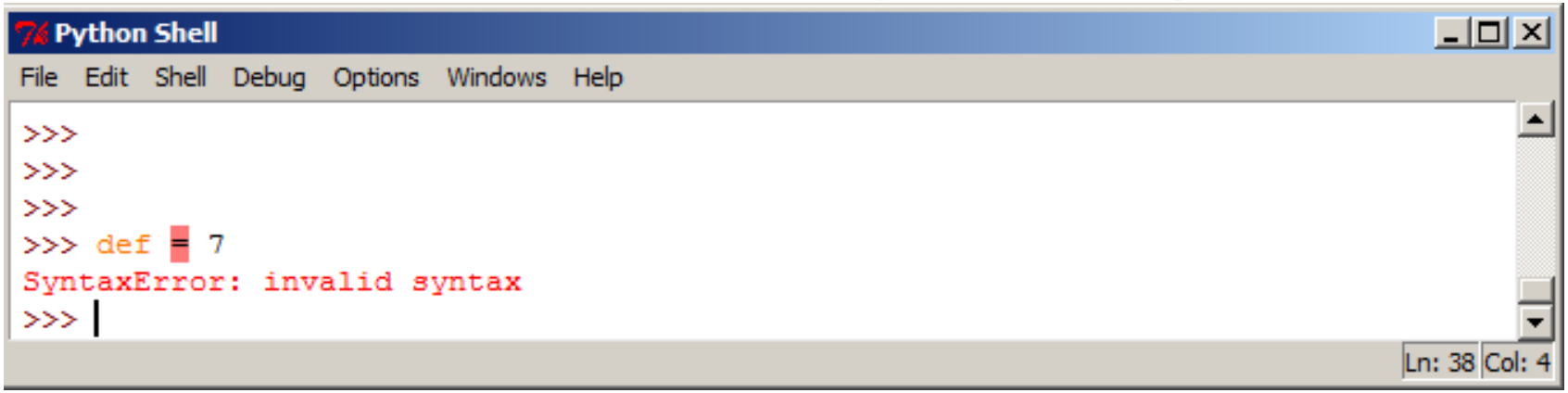
```
>>>help('keywords')

Here is a list of the Python keywords. Enter any keyword to get
more help.
```

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

# Python Naming Conventions

## Key Words in Python




**"SyntaxError: invalid syntax..." is Python's ways of saying, "Hey, you don't know what you are talking about and neither do I! Speak Python!"**

# Python Naming Conventions

- Variable names can contain **only letters, numbers, and underscores**. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable **`message_1`** but not **`1_message`**.
- **Spaces** are not allowed in variable names, but underscores can be used to separate words in variable names. For example, **`greeting_message`** works, but **`greeting message`** will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word **`print`**.
- Variable names should be short but descriptive. For example, **`name`** is better than **`n`**, **`student_name`** is better than **`s_n`**, and **`name_length`** is better than **`length_of_persons_name`**.
- Be careful when using the lowercase letter **`l`** and the uppercase letter **`O`** because they could be confused with the numbers **`1`** and **`0`**.

# Python Naming Conventions

## Variables Naming Rules

- You cannot use one of Python's key words as a variable name.
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name ItemsOrdered is not the same as itemsordered.

# Variables

## Strings in Python

When you type a string into Python, you do so by preceding it with quotes. Whether these quotes are single ( ' ), double( " ), or triple( " " " ) depends on what you are trying to accomplish.

```
>>> "This is a string using a double quote"
'This is a string using a double quote'
>>> 'This is a string with a single quote'
'This is a string with a single quote'
>>> """This string has three quotes
look at what it can do!"""
'This string has three quotes\nlook at what it can do!'
```

# Variables

```
>>> boilerplate = """
    #===(")===#===(*)===#===(")===#
    Egregious Response Generator
    Version '0.1'
    "FiliBuster" technologies inc.
    #===(")===#===(*)===#===(")===#
    """
>>> print(boilerplate)

    #===(")===#===(*)===#===(")===#
    Egregious Response Generator
    Version '0.1'
    "FiliBuster" technologies inc.
    #===(")===#===(*)===#===(")===#

>>>
```

# Variables

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
When unpacking, you may sometimes want to discard certain
values
========== RESTART ==========
s = 'Hello'
>>> _, b, c, d, e = s
>>> a
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

# Variables

## Casting String To Integer

```
>>> int('99')
99
>>> int('-29')
-29
>>> int('99 bottles of milk!')
ValueError: invalid literal for int() with base 10: '99
bottles of milk!'
>>> int(' ')
ValueError: invalid literal for int() with base 10: ' '
>>> int('98.6')
ValueError: invalid literal for int() with base 10:
'98.6'
>>>
```

# Variables

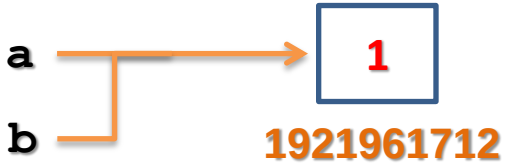
## Casting String To Float

```
>>> float(98)
98.0
>>> float('98')
98.0
>>> float('99 bottles of milk!')
ValueError: could not convert string to float: '99
bottles of milk!'
>>> float('99.3')
99.3
>>> float(' ')
ValueError: could not convert string to float:
```

# Variables

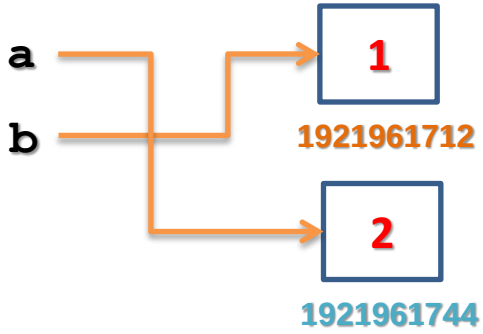
Each data (or object) in Python is assigned a unique identifier (basically, an integer) which can be accessed by the **id()** function. Having unique identifiers, Python manages memory space such that multiple occurrences of the same data are stored only once whenever possible. For example:

```
>>> a = 1
>>> b = 1
>>> id(1)
1921961712
>>> id(a)
1921961712
>>> id(b)
1921961712
```



# Variables

```
>>> a = 1
>>> b = 1
>>> id(1)
1921961712
>>> id(a)
1921961712
>>> id(b)
1921961712
>>> a = 2
>>> b
1
>>> id(a)
1921961744
>>> id(b)
1921961712
```



# Variables

```
>>> a = 's'
>>> b = 's'
>>> id(a)
2197299367088
>>> id(b)
2197299367088
>>> a == b
True
```

```
>>> a = 'serdar'
>>> b = 'serdar'
>>> id(a)
2197335546608
>>> id(b)
2197335546608
id('serdar')
2197335546608
```

# Everything Is an Object

```
from objbrowser import browse
a = 16
browse(locals())
```

| name | path | summary | unicode | repr | type name |
|---|---|---|---|---|---|
| _spec_ | _spec_ | None | None | None | NoneType |
| a | a | 16 | 16 | 16 | int |
| _abs_ | a._abs_ | | <method-wrapper '__abs__' of in... | <method-wrapper '__abs__' of in... | method-wrapp |
| _add_ | a._add_ | | <method-wrapper '__add__' of in... | <method-wrapper '__add__' of in... | method-wrapp |
| _and_ | a._and_ | | <method-wrapper '__and__' of in... | <method-wrapper '__and__' of in... | method-wrapp |
| _bool_ | a._bool_ | | <method-wrapper '__bool__' of i... | <method-wrapper '__bool__' of i... | method-wrapp |
| _ceil_ | a._ceil_ | | <built-in method __ceil__ of int o... | <built-in method __ceil__ of int o... | builtin_functior |
| _class_ | a._class_ | | <class 'int'> | <class 'int'> | type |
| _delattr_ | a._delattr_ | | <method-wrapper '__delattr__' of... | <method-wrapper '__delattr__' of... | method-wrapp |
| _dir_ | a._dir_ | | <built-in method __dir__ of int o... | <built-in method __dir__ of int o... | builtin_functior |
| _divmod_ | a._divmod_ | | <method-wrapper '__divmod__' ... | <method-wrapper '__divmod__' ... | method-wrapp |
| _doc_ | a._doc_ | int([x]) -> integer↵int(x, base=1... | int([x]) -> integer↵int(x, base=1... | "int([x]) -> integer\nint(x, base=... | str |
| _eq_ | a._eq_ | | <method-wrapper '__eq__' of int ... | <method-wrapper '__eq__' of int ... | method-wrapp |
| _float_ | a._float_ | | <method-wrapper '__float__' of i... | <method-wrapper '__float__' of i... | method-wrapp |
| _floor_ | a._floor_ | | <built-in method __floor__ of int ... | <built-in method __floor__ of int ... | builtin_functior |
| _floordiv_ | a._floordiv_ | | <method-wrapper '__floordiv__' ... | <method-wrapper '__floordiv__' ... | method-wrapp |
| _format_ | a._format_ | | <built-in method __format__ of i... | <built-in method __format__ of i... | builtin_functior |
| _ge_ | a._ge_ | | <method-wrapper '__ge__' of int ... | <method-wrapper '__ge__' of int ... | method-wrapp |
| _getattrib... | a._getattribute_ | | <method-wrapper '__getattribute... | <method-wrapper '__getattribute... | method-wrapp |
| _getnewar... | a._getnewargs_ | | <built-in method __getnewargs_... | <built-in method __getnewargs_... | builtin_functior |

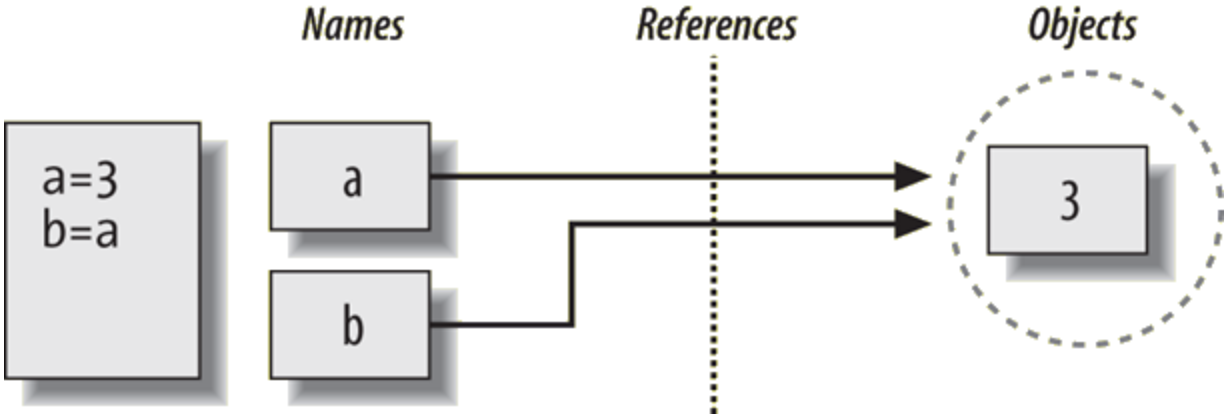
Details

path __file__

# Variables

Python's **garbage collection** is based mainly upon ***reference counters***, however, it also has a component that detects and reclaims objects **with *cyclic references*** in time. This component can be disabled if you're sure that your code doesn't create cycles, but it is enabled by default. ,

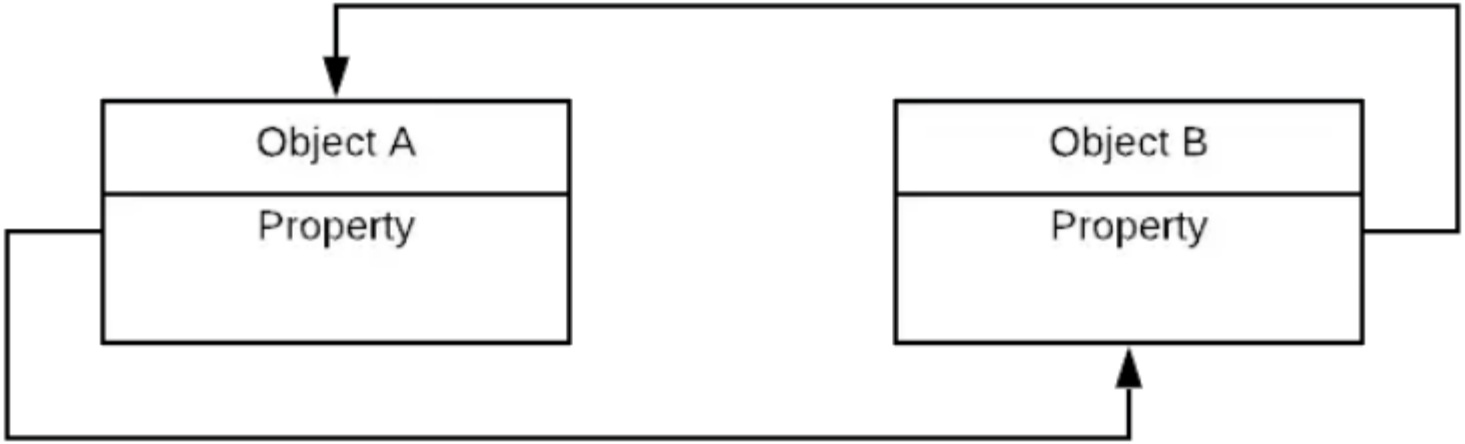
```
>>> a = 3
>>> b = a
```




the variables a and b wind up referencing the same object

# Variables

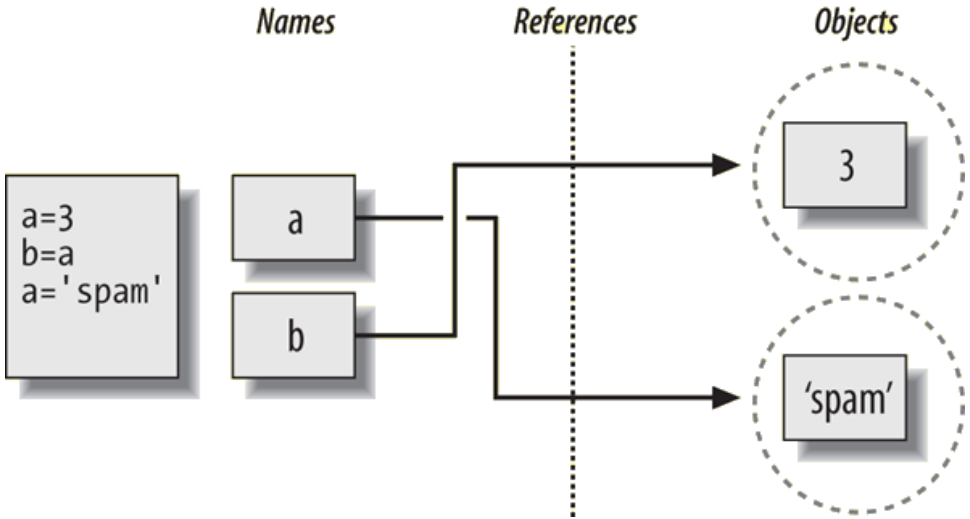
A circular reference is a cyclic dependency in Python. Python's garbage collector runs automatically when a program exits. It will try to free unused objects by removing them from the program's memory. If you have a circular reference, the garbage collector will run infinitely until your program eventually crashes.



# Variables

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

After running the assignment **a = 'spam'**. Variable a references the new object (i.e., piece of memory) created by running the literal expression 'spam', but variable b still refers to the original object 3. Because this assignment is not an in-place change to the object 3, it changes only variable **a**, not **b**.




in Python variables are always pointers to objects, not labels of changeable memory areas: setting a variable to a new value does not alter the original object, but rather causes the variable to reference an entirely different object.

# Variables

**For more details on Python's cycle detector, see the documentation for the gc module in Python's library manual.**

## gc — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in gc.garbage for inspection.

* Garbage Collection is the process by which unused objects are deleted to reclaim memory space
* Invented by John McCarthy around 1959 to solve problems in Lisp

# Operators

Operators do something, like add, multiply, divide, subtract, or compare. Notice that we already used the = sign to assign identifiers. So, we can't use the equal sign as an operator. Instead we must use == to mean; "equals." What other useful things can be done with operators?

In the Python shell type:
"words words words words words"
Hit enter

Now type:

```
"words " * 10
```

Hit enter. ???

# Operators

Arithmetic operators we will use:

| | |
|---|---|
| + | addition |
| - | subtraction/negation |
| * | multiplication |
| / | division |
| % | modulus, a.k.a. remainder |
| ** | exponentiation |

precedence: Order in which operations are computed.
* / % ** have a higher precedence than + -

1 + 3 * 4 is 13

Parentheses can be used to force a certain order of evaluation.
(1 + 3) * 4 is 16

# Operators

```
>>>
>>> 1.0023 - 1.0567
>>>
>>> 1000.0023 - 1000.0567
>>>
>>> a = 1234.567
>>> b = 45.67834
>>> c = 45.67834
>>> d = (a + b) + c
>>> e = a + (b + c)
>>> print(d)
>>> print(e)
```

**Try these..**

**Floating point calculations !!!**

# Operators
# Integer Division

When we divide integers with // , the quotient is also an integer.

```
>>> 3//3
1
>>> 3//2
1
>>> 5//8
0
>>> 8//5
1
```

The % operator computes the remainder from a division of integers.

```
>>> 8%5
3
>>> 3%2
1
>>> 3%3
0
```

# Operators

We also have comparison operators for…
you guessed it; comparing things!
Here are some of those:

| | |
|---|---|
| **>** | **for greater than** |
| **<** | **for less than** |
| **<=** | **less than or equal to** |
| **>=** | **greater than or equal to** |
| **= =** | **equal to** |
| **! =** | **not equal to** |

Remember that we already used the = symbol to define or tell the computer the meaning of our variables. So we don't confuse our little computer, we must use = = to express the traditional meaning of "**equal to**."

# Operators

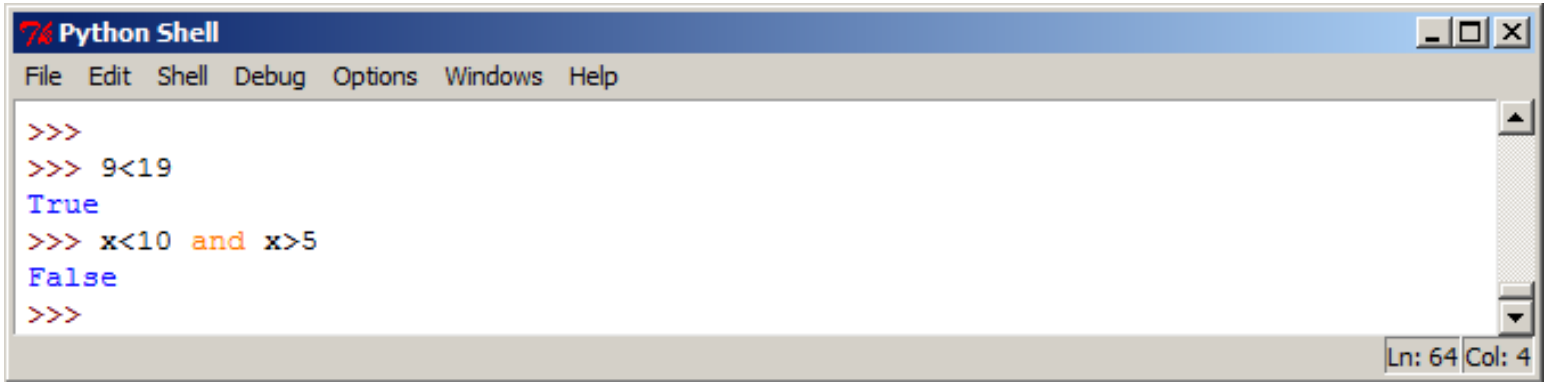
**Boolean** is the type of data that represents the answer to questions like `9 < 19`. The words "**`and`**, **`or`**," and "**`not`**" are logical operators. A simple condition is a comparison that only uses two values:

**`9<19`**

A compound condition is a comparison using more than two values:

**`x < 10`** `and` **`x>5`**

These logical operators generally mean the same thing in programming as they do in English.



# Operators

```
>>> 1 < 2     # Less than
True
>>> 2.0 >= 1 # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0       # Equal value
True
>>> 2.0 != 2.0       # Not equal value
False
>>> X = 2
>>> Y = 4
>>> Z = 6
>>> X < Y < Z        # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

# Operators

```
>>> int(False)
>>> ?
>>> int(True)
>>> ?
>>> not 0
>>> ?
>>> not 1
>>> ?
>>> not " "
>>> ?
>>> not "This is some text"
>>> ?
>>> float(True)
>>> ?
```

**Try these..**

**What is the answer?**

# 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 = 50
>>> print ("You have", 65 - age, "years until
retirement")
Output:
  Hello, world!
  You have 15 years until retirement
```

# The `print` Function

The standard output function print displays its arguments on the console. This function allows a variable number of arguments. Python automatically runs the `str` function on each argument to obtain its string representation and separates each string with a space before output. By default, print terminates its output with a newline.

```
Line 1| message = "Hello from Python Programing Course"
Line 2| print(mesage)

Traceback (most recent call last):
 File "hello_world.py", line 2, in <module>
 print(mesage)
 NameError: name 'mesage' is not defined
```

# print()

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

```
>>> print('This is a string using a single quote!')
This is a string using a single quote!
>>> print("This is a string using 'a double' quote!")
This is a string using 'a double' quote!
>>> print("""This string has three "quotes\n Look" at what
it can do!""")
This string has three "quotes
Look" at what it can do!
>>> print("I said, "Don't do it")
SyntaxError: invalid character in identifier
>>> print("I said, \"Don't do it\"")
I said, "Don't do it"
```

When Python saw the backslash (\), or escape character, it knew to treat the double quote as a character, and not as a data type indicator

# Variables

```
>>> print("Using double quotes")
>>> print('Using single quotes')
>>> print("Mentioning the word ‘Python’ by quoting it")
>>> print("Embedding a\nline break with \\n")
>>> print("""Embedding a
line break with triple quotes""")
```

Output:

```
Using double quotes
Using single quotes
Mentioning the word 'Python' by quoting it
Embedding a
line break with \n
Embedding a
line break with triple quotes
>>>
```

# Variables

```
>>> print("Example Heading\n\nFollowed by a line\nor two
of text\n \\tName\n\tRace\n\tGender\nDon\'t forget to
escape \'\\\'.")

Example Heading

Followed by a line
or two of text
        Name
        Race
        Gender
Don't forget to escape '\'.
>>>
```

# print()

## Basic Escape Sequences

| Escape sequence | Character represented |
|---|---|
| \' | Single-quote character |
| \" | Double-quote character |
| \\ | Backslash character |
| \a | Bell character |
| \b | Backspace character |
| \f | Formfeed character |
| \n | Newline character |
| \r | Carriage return character (not the same as \n) |
| \t | Tab character |
| \v | Vertical tab character |

# print()

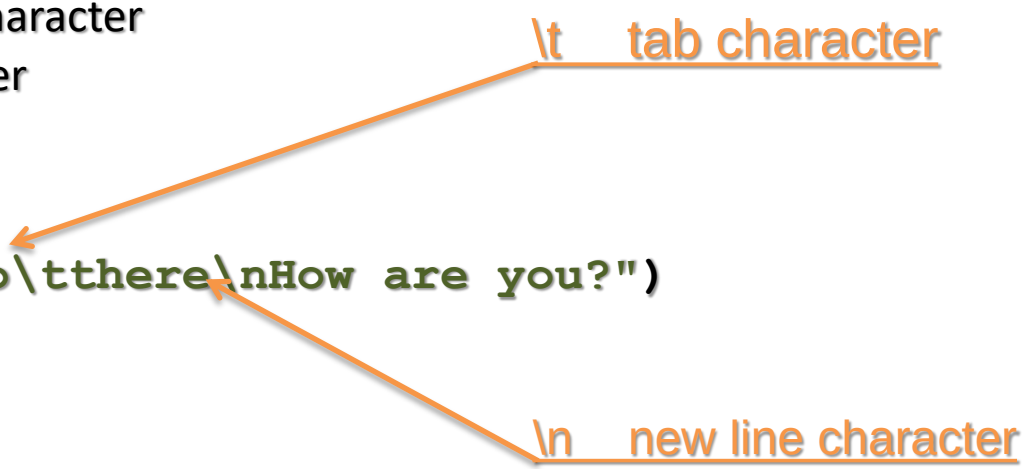
## Basic Escape Sequences

A string can represent characters by preceding them with a backslash.

| | |
|---|---|
| \t | tab character |
| \n | new line character |
| \" | quotation mark character |
| \\ | backslash character |

Example:

```
>>> print("Hello\tthere\nHow are you?")
Hello   there
How are you?
```



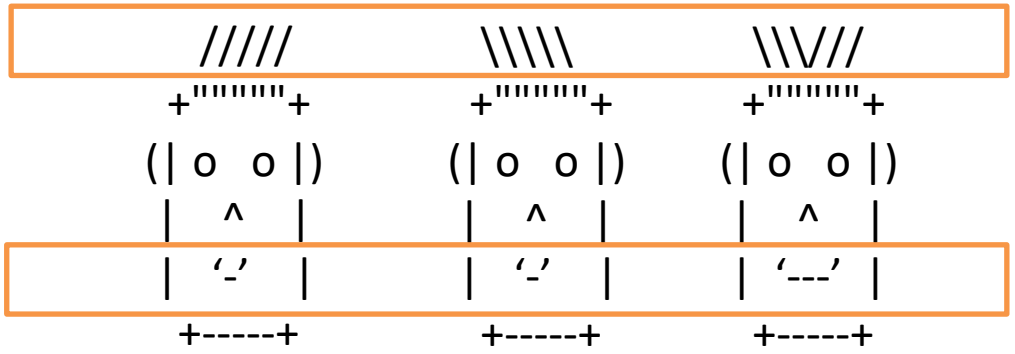
# print()

## Basic Escape Sequences

```
>>> start = 'Ya' * 3 + '\n'
>>> middle = 'Şa' * 3 + '\n'
>>> end = 'En Büyük….'
>>> print(start + middle + end)
>>> ?
```

# print()

**Write a program that prints a face similar to the following:**



# print()

Ulaşım Masrafı

1. Cumartesi - Pazar günleri çalışmıyoruz.
2. Dolayısıyla ayda 22 gün çalışıyoruz.
3. Evden işe gitmek için kullandığımız aracın ücreti 2.5 TL
4. İşten eve dönmek için kullandığımız aracların ücreti 3.4 TL

```
gün = 22
gidiş_ücreti = 2.5
dönüş_ücreti = 3.4
masraf = gün * (gidiş_ücreti + dönüş_ücreti)
print(masraf)
```

# print()

```
gün = 22
gidiş_ücreti = 2.5
dönüş_ücreti = 3.4
masraf = gün * (gidiş_ücreti + dönüş_ücreti)

print("-"*30)
print("çalışılan gün sayısı\t:", gün)
print("işe gidiş ücreti\t:", gidiş_ücreti)
print("işten dönüş ücreti\t:", dönüş_ücreti)
print("-"*30)
print("AYLIK ULAŞIM MASRAFI\t:", masraf)
```

# print()

```
>>> "serdar" + "aritan"
'serdararitan'
>>> "serdar" + " " + "aritan"
'serdar aritan'
>>> "serdar" + "." + "aritan"
'serdar.aritan'
```

Using a Format Specifier to Populate a String

```
>>> "%s %s %10s" % ("Serdar" , "Aritan", "Hacettepe")
'Serdar Aritan  Hacettepe'

>>> "%s %s %20s" % ("Serdar" , "Aritan", "Hacettepe")
'Serdar Aritan           Hacettepe'
```

# print()

## Format Specifier

```
>>> num = 1 / 3.0
>>> num                          # Auto-echoes
0.3333333333333333
>>> print(num)                   # Print explicitly
0.3333333333333333
>>> '%e' % num                   # String formatting expression
'3.333333e-01'
>>> '%4.2f' % num                # Alternative floating-point
format
'0.33'
>>> '{0:4.2f}'.format(num)            # String formatting method
'0.33'
```

# print()

## Format Specifier

| Format String | Sample Output | Comments |
|---|---|---|
| "%d" | 24 | Use d with an integer. |
| "%5d" | ␣␣␣24 | Spaces are added so that the field width is 5. |
| "%05d" | 00024 | If you add 0 before the field width, zeroes are added instead of spaces. |
| "Quantity:%5d" | Quantity:␣␣␣24 | Characters inside a format string but outside a format specifier appear in the output. |
| "%f" | 1.219977 | Use f with a floating-point number. |
| "%.2f" | 1.22 | Prints two digits after the decimal point. |
| "%7.2f" | ␣␣␣1.22 | Spaces are added so that the field width is 7. |
| "%s" | Hello | Use s with a string. |
| "%d %.2f" | 24 1.22 | You can format multiple values at once. |
| "%9s" | ␣␣␣␣Hello | Strings are right-justified by default. |
| "%-9s" | Hello␣␣␣␣ | Use a negative field width to left-justify. |
| "%d%%" | 24% | To add a percent sign to the output, use %%. |

# print()

## Format Specifier

```
>>> myscore = 1000
>>> message = 'I scored %s points'
>>> print(message % myscore)
I scored 1000 points
```

TRY These..

I scored 1000 points, how about you Serdar

Your height 1.80 cm

Your name [ Serdar  Arıtan ]

# print()

## F-Strings Format Specifier

### Strings in Python

Python 3.6 introduced an alternative, more compact, way to build string expressions. An **f-string** consists of the character `f` (or `F`) following by a special kind of string literal called a **formatted string literal**. Formatted string literals contain both sequences of characters (like other string literals) and expressions bracketed by **curly braces**. These expressions are evaluated at runtime and automatically converted to strings. The code.

```
print(f'{int(num*fraction)} is {fraction*100}% of {num}')
```

# print()

## F-Strings Format Specifier

```
>>> msg = 'hello world'
>>> 'msg: %s' % msg
'msg: hello world‘

>> msg = 'hello world'
>>> 'msg: {}'.format(msg)
'msg: hello world‘

>>> msg = 'hello world'
>>> f'msg: {msg}'
'msg: hello world'
```

# print()

## F-Strings Format Specifier

```
>>> book = "The dog guide"
>>> num_pages = 124
>>> f"The book {book} has {num_pages} pages"
'The book The dog guide has 124 pages'
>>> F"The book {book} has {num_pages} pages"
'The book The dog guide has 124 pages'
>>> print(Fr"The book {book} has {num_pages} pages\n")
The book The dog guide has 124 pages\n
>>> print(FR"The book {book} has {num_pages} pages\n")
The book The dog guide has 124 pages\n
>>> print(f"The book {book} has {num_pages} pages\n")
The book The dog guide has 124 pages‘
↓
>>>
```

# print()

## F-Strings Format Specifier

```
>>> f"4 * 4 is {4 * 4}"
'4 * 4 is 16'
>>>

>>> n = 4
>>> f"4 * 4 is {n * n}"
'4 * 4 is 16'

>>> def magic_number():
...     return 42
>>> f"{magic_number() = }"
'magic_number() = 42'
>>> f"{magic_number()}"
'42'
```

# input()

**input** : Reads a number from user input.

You can assign (store) the result of input into a variable.

**Example**:

```
age = input("How old are you? ")
print ("Your age is", age)
print ("You have", 65 - age, "years until retirement")
```

Output:

```
How old are you? 53
Your age is 53
You have 12 years until retirement
```

**What wrong with this?**

**Why it does not run?**

# input()

**input** : Reads a number from user input.

You can assign (store) the result of input into a variable.

**Example**:

```
age = input("How old are you? ")
print ("Your age is", age)
print ("You have", 65 - int(age), "years until retirement”)
```

Output:

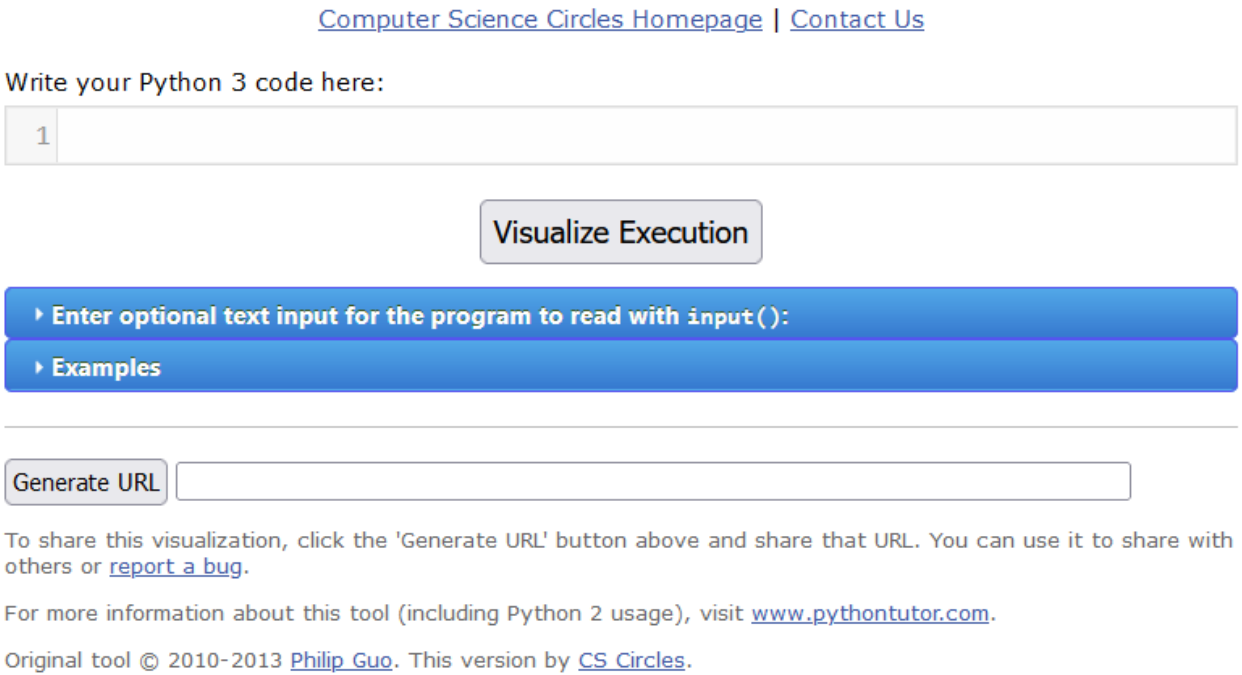
```
How old are you? 55
Your age is 55
You have 10 years until retirement
```

# input()

```
# Calculate the area and circumference of a circle from its radius
# Step 1: Prompt for a radius.
# Step 2: Apply the area formula.
# Step 3: Print out the results.
import math
radius_str = input("Enter the radius of your circle: ")
radius_int = int(radius_str)
circumference = 2 * math.pi * radius_int
area = math.pi * (radius_int ** 2)
print ("The cirumference is:",circumference, \
", and the area is:",area)
>>>
Enter the radius of your circle: 20
The cirumference is: 125.6637061435917 , and the area is:
1256.6370614359173
```

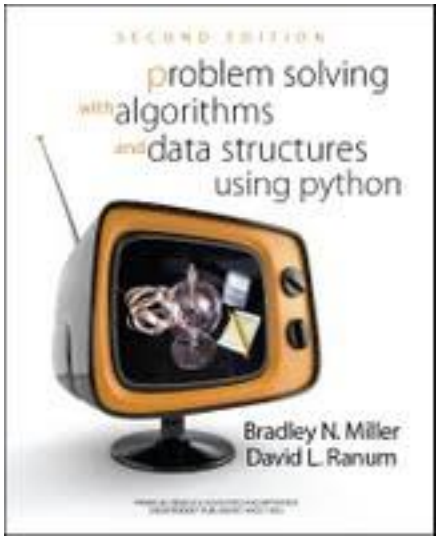
# Resources

https://cscircles.cemc.uwaterloo.ca/visualize



# Online Resources

https://runestone.academy/ns/books/published/pythonds/index.html




https://www.youtube.com/user/gjenkinslbcc

# Online Resources

https://runestone.academy/ns/books/published/pythonds/Introduction/GettingStartedwithData.html

## 1.8. Getting Started with Data

We stated above that Python supports the object-oriented programming paradigm. This means that Python considers data to be the focal point of the problem-solving process. In Python, as well as in any other object-oriented programming language, we define a **class** to be a description of what the data look like (the state) and what the data can do (the behavior). Classes are analogous to abstract data types because a user of a class only sees the state and behavior of a data item. Data items are called **objects** in the object-oriented paradigm. An object is an instance of a class.

### 1.8.1. Built-in Atomic Data Types

We will begin our review by considering the atomic data types. Python has two main built-in numeric classes that implement the integer and floating point data types. These Python classes are called `int` and `float`. The standard arithmetic operations, +, -, *, /, and ** (exponentiation), can be used with parentheses forcing the order of operations away from normal operator precedence. Other very useful operations are the remainder (modulo) operator, %, and integer division, //. Note that when two integers are divided, the result is a floating point. The integer division operator returns the integer portion of the quotient by truncating any fractional part.

# Vocabulary Review

**Boolean** data type is referring to two possible values; True or False.

**Comparison operators** are for comparing things, (<, >, ==, !=, etc.).

**Compound condition** is a comparison using more than two values, (x<10 and x>5)

**Conditional expressions** (also called Boolean expressions), are based on the condition that something is either true or false.

**Delimiters** quotation marks to tell the computer we are entering a string.

# Vocabulary Review

**Float** non-whole numbers like decimals.

**Floating point** numbers with a decimal point

**Identifiers** are names

**Integer** a complete number as opposed to part of a number like ½

**Logical operators** the words “and, or,” and “not”

**Operators** do something, like add, multiply, divide, subtract, or compare.