BASIC OPERATORS

`+` *addition*
`-` *subtraction*
`*` *multiplication*
`/` *division*
`**` *power/exponent*

`//` *integer division*
Ex: 7//3 = 2 *division without remainder*

`%` *modulus*
Ex: 7%3 = 1 *remainder from division*

`from math import *`

Allows us to use more math functions:

```
sqrt(x)
cos(x), sin(x), tan(x)
acos(x), asin(x), atan(x)
log(x), log10(x)
exp(x)
```
this is e^x

NOTE: trig functions take in radian values, not degrees, so you might need to convert

VARIABLES

`<variable name> = <value to assign>`

TYPES
Integers (Ex: 1, 2, 100, 0, -20)
Floats (Ex: 1.01, 2.0, 3.14, -20.03)
Booleans (`True, False`)
Strings (stuff in quotes; Ex: 'Howdy!', "72.3")

Do not include (-) or (') in a variable name

CONVERTING BETWEEN TYPES

`int(value)` converts value to integer
Ex: int("2") = 2
int(4.9) = 4 decimal portion drops off
int("3.0") → invalid literal; doesn't convert

`float(value)` converts value to a float
Ex: float(5) = 5.0
float("3.14") = 3.14
float("12") = 12.0

`str(value)` converts value to a string
Ex: str(2.5) = "2.5"
str(1) = "1"

`bool(value)` converts value to `True`; except 0, 0.0, and *empty* "", [] convert to `False`

`input()`

All input comes in as a string, so if we want to use the inputted value in calculations, we need to convert to either an integer or float.

Ex: `age=int(input("Enter your age:"))`

COMPARISON OPERATORS

It allows us to compare two values; results in a Boolean value.

`==` *Equality*
`!=` *Inequality*
`<, <=` *less than, less than or equal to*
`>, >=` *greater than, greater than or equal to*

BOOLEAN OPERATORS

`not, and, or`

A `and` B (True if both A and B are true)
A `or` B (True if either A or B is true, or if both are true)
`not` A ("reverses" A; True if A is False, False if A is True)

CONDITIONAL STATEMENTS

```
if <condition>:
    <do this>
```

Code indented inside happens if the condition is True

if-else

Use when there are 2 possibilities

```
if (condition):
    <do this if condition is True>
else:
    <do this when condition is False>
```

if-elif-else

Use when there are more than 2 alternative paths

```
if (condition1):
    <do this if condition1 is True>
elif (condition2):
    <do this if condition2 is True>
…
else:
   <do this is all above conditions are False>
```

NOTE: you can nest conditional statements
Ex:
```
num = int(input("Enter a #:"))
If num > 5:
   print("Greater than 5")
      If num <= 47:
        print("Between 5 and 47")
      else:
        print("Greater than 47")
```

WHILE LOOP

Repeats the indented code until the condition is False.

```
while <condition>:
    <do this>
```

If the condition is True, all the indented code is run. After the code is run, the condition is checked again; if a condition is still **true**, the code is run again, and so on. If False the code is skipped.

```
print("guess number between 1 and 0.")
secret_number = 7
User = int(input("guess number"))
While User != secret_number:
  print ("no.try again")
  User = int(input("guess number"))
print("correct")
```

FOR LOOP

Repeats the indented code a specific number of times whether the condition is true or false.

```
i = 0
For i in range < condition >:
       < do this >
```

The for loop is used when you have a known number of iterations or when you want to iterate through a specific, known set of items.

```
List = [a,b,c,d]
For i in range List:
       < do somthing >
```

Comments

Include comments within your code to clarify your code, computation, and purpose.

TESTS

Tests should be thought of before you write any code.

Typical cases

You use this case to test for common possible errors within your code. This is what the user should give you.

Edge or Corner cases

You use this case to counter special error cases that would be less common to occur within your code. What happens if the user messes up. (invalid inputs)

Errors

Referring to Logical (incorrect result), Runtime (execution) or Syntax (language/compiler) mistakes within the code.

More list Commands

Used to manipulate or interact with lists.

Sequence type commands

`len(list)` length of the list.
`min(list)` minimum value in the list.
`max(list)` maximum in the list.
`sum(list)` sum of all variables in the list.

`list.index(val)` finds the index of the first element in the list whose value matches val.

`list.count(val)` finds the number of occurrences of the val in the list.

`List[start: end]` allows you to collect certain values in a list.

LISTS

We can use lists to store several values of the same type.

```
<list name> = [ ]
```

[ ] indicates a list, and individual elements are separated by commas.

ACCESSING ELEMENTS IN A LIST

```
<list name>[<element number>]
```

The first item in a list is at element 0, the second is at element 1, and so on.

Ex:
```
grades = [92, 84, 89, 96]
print(grades[0])   outputs 92
print(grades[2])   outputs 89
```

OTHER LIST COMMANDS

`len(x)` length, returns the number of elements in list x

`<list name>.append(<element to add>)` adds an element to the end of the list

`del <list name>[<element number>]` removes the item at that position

`<list name>.pop()` removes the last item in the list

Ex:
```
grades = [92, 84, 89, 96]
grades.append(79)
                adds 79 to end of list
del grades[1]     removes 84
grades.pop()      removes 79
```