Variables & numbers

- Video: Learning Python (2018 edition) 01: Your first programs (Variables, numbers, strings & casting)

Basic calculations

Python supports all the basic arithmetic calculations

```
print( 2 + 2 )          # addition
print( 1.5 + 2.25 )     # addition
print( 7 - 2 )          # subtraction
print( 3 * 4 )          # multiplication
print( 10 / 2 )         # division
print( 4 ** 3 )         # exponent
```

A note about division & modulus

```
print( 13 / 5 )   # Real number divison
print( 13 // 5 )  # Integer divison
print( 13 % 5 )   # Modulus
```

Variables

Variables are just a named memory location

Defining a variable in Python is as simple as assigning a value to a name.

```
a = 10
```

- Names must start with an alpha character or underscore, but may then contain numeric characters.
- Names should be meaningful. Establish good habbits early. It should be obvious from the name of the variable what it's purpose is.
- Python's preferred practice is to separate_words_with_underscores rather than using camelCase like otherLanguages.
- Be warned variable names are case sensitive. `Variable` is not the same as `variable`.

```
var = 10
print(Var) # Will not work!
```

Using variables in calculations

Calculations can be assigned on a right goes into left basis.

```
val = 5 + 3
print(val)
```

Variables can be used as part of a calculation as well

```
a = 5 + 3
b = a * 4
c = a - b
print( c ** a )
```

An example using the basic mathematical operators would be...

```
a = 100
b = 6
c = a + b           # addition          ... c == 106
d = a - b           # subtraction       ... c == 94
e = a * b           # multiplication    ... c == 400
f = a / b           # division          ... c == 16.66667
g = a // b          # integer division  ... c == 16 (how many times does 6 go into 100)
h = a % b           # modulus remainder ... c == 4 (ie: remainder of 100 divided by 6)
i = a ** b          # exponent          ... c == 1000000000000 (ie: 10^6)
print(c,d,e,f,g,h,i)
```

Integers vs real numbers

Python has two types of numbers: integers and floats. Integers are "whole numbers" without decimals, floats are the name given to numbers that contain decimals.

To denote a variable to be a real number simply adding a decimal element.

```
print(type(13))
print(type(13.0))
```

To convert a real number to integer, use the `int()` command to truncate, or `round()` to round.

```
a = 13.6
b = int(a)
c = round(a)
print(a,b,c)
```

To convert an integer to real, use the `float()` command.

```
a = 13
b = float(a)
print(a,b)
```

Other numerical functions

```
import math
answer = math.pi                # π = 3.141592
answer = math.e                 # the natural number, e = 2.718281
answer = math.sqrt(100)         # Square root
answer = math.gcd(104,64)       # Greatest common divisor
answer = math.log(1024,2)       # Log of base 2
answer = math.hypot(6,8)        # Hypothenus of triangle with sides 6, 8
answer = math.cos( angle )      # Cosine of angle (radians)
answer = math.sin( angle )      # Sine of angle (radians)
answer = math.tan( angle )      # Tangent of angle (radians)
answer = math.acos( adj/hypot ) # Arc-cosine in radians
answer = math.asin( opp/hypot ) # Arc-sine in radians
answer = math.atan( opp/adj )   # Arc-tan in radians
answer = math.degrees( rad )    # Convert radians to degrees
answer = math.radians( deg )    # Convert degrees to radians
answer = abs( val )             # Absolute value

import random
num = random.randint(0,100)     # Random number between 0 and 99 inclusive
```

Example: How long is the hypotenuse of a triangle if the adjacent side is 20, and the angle is 45 degrees?

```
import math
adjacent = 20.0
angle = 45.0
hypotenuse = adjacent / math.cos( math.radians( angle ))
print(hypotenuse)       # prints 28.284...
```

Problem set

The following questions assume you will use variables as the inputs into the problem, so the problems can re-caclulate solutions by changing the value assigned to the variable. You should also print the given information in your answer.

1. For any given number, extract the 10s digit. For example, `The tens digit in 1234 is 3.`
2. Area of a right angled triangle calculator. Given values for `base` and `height`, print the area.
3. For any two digit number, swap the position of the digits. For instance, `79` becomes `97`.
4. For any three digit number, print the sum of the three digits. For instance `273` becomes `12` (2+7+3)
5. For any given year, print the century that year belongs to. Remember that 1999 and 2000 were the 20th century, whereas 2001 was the beginning of the 21st century.
6. Given a number representing the number of seconds since midnight, print the time in 24hour clock format. For example `70500` seconds should print a time of `19:35`.
7. Area of a non-right angled triangle calculator. Given values for length `a`, length `b` and angle in degrees `C`, return the area of the triangle (remember you will have to convert degrees to radians first).
8. For any given values for `a`, `b` and `c`, will provide the solutions to the quadratic formula (you may assume both solutions are required). Be careful with your order of precedence. Here is an example solution set for testing: If `y=2x^2-4x-10` then the solutions are `3.44949` and `-1.44949`.

Printing & input

To print text to screen:

```
print("Hello world!")
```

To print a variable to screen:

```
name = "Han Solo"
record = 12
print(f"{name} completed the Kessel run in {record} parsecs")
# notice the 'f' in front of the first set of quotes
```

To ask the user for input

```
name = input("What is your name? ")                         ## Input saved as text string
num = int(input("Type an integer between 1 and 100: "))     ## Input saved as an integer
double = num * 2
print(f"Hello {name}, double your number is {double}")
```

Sometimes you need to format the variables you are printing.

```
val = 12
print( f"With leading spaces to make it 4 characters wide is {val:4}" ) # prints '  12'
print( f"With leading zeros to make it 4 characters wide is {val:04}" ) # prints '0012'
val = 3.14
print( f"To 2 decimal places is {val:.2f}" ) # prints '3.14'
print( f"To 5 decimal places is {val:.5f}" ) # prints '3.14000'
print( f"With spaces to make it 8 characters wide with 3 decimal places is {val:8.3f}" )
print( f"With zeros to make it 8 characters wide with 3 decimal places is {val:08.3f}" )
```

Strings & casting

- If you haven't already watched this video, I suggest you do so now. Learning Python (2018 edition) 01: Your first programs (Variables, numbers, strings & casting)

String variables

A text variable is known as a "string" (a string of characters).

- The starting and end point of the text is denoted by a set of quotes or double quotes.

```
a = "This is a string"
b = 'This is a string'
c = "This is not a string'  # different opening and closing quote symbols
```

Strings can be concatenated (joined) together with the `+` operator.

```
a = "Hello"
b = "world!"
c = a + " " + b
print(c)
```

Printing & f-strings

> f-Strings require a minimum of Python version 3.6

A convenient way of inserting the content of other variables is an f-string.

- Preceed the opening quotes with the `f` character
- Wrap variable names with `{}` characters

```
a = 10
print(f"The value of a is {a}")
```

F-Strings have a variety of methods for formatting the value being inserted into a string.

- Decimal places

```
val = 12.3
print(f'{val:.2f}')     # 2 decimal places, eg 12.30
print(f'{val:.5f}')     # 5 decimal places, 12.30000
```

- Width

```
val = 12.3
print(f'{val:10}')      # 10 characters wide, eg "      12.3"
print(f'{val:10.5f}')   # 10 characters wide, 5 decimal places, eg "  12.30000"
```

- Left and right justify (strings default to left justify, numbers default to right justify)

```
val = 12.3
txt = "Hello"
print(f'{val:<10}')      # 10 characters wide left justify, eg "12.3      "
print(f'{txt:>10}')      # 10 characters wide right justify, eg "     Hello"
```

- Hexadeciaml notation

```
num = 255
print(f"{num:x}")       # Hexademcimal value, eg "ff"
```

Credit: http://zetcode.com/python/fstring/

Inputting strings

We can prompt the user to input data and store it into a string using the `input()` command.

```
person_name = input("What is your name?")
print(f"Hello {person_name}")
```

Sub strings

To extract parts of a string we use a set of square brackets after our variable name.

```
name = "Luke Skywalker"
print( name[5:] )       # Skywalker
print( name[5:8] )      # Sky
print( name[:4] )       # Luke
print( name[-9:] )      # Skywalker
```

- String positions start from zero
- The bracket notation works as `[ after_this_position : up_to_this_position ]`
- When asking for a range of characters, Python will give you a substring that includes the starting position number, but not including the end position number.

String functions

Changing strings

```
original_text = "To infinity and beyond!"
new_text = original_text.lower()              ## == "to infinity and beyond!"
```

```
new_text = original_text.upper()              ## == "TO INFINITY AND BEYOND!"
new_text = original_text.title()              ## == "To Infinity And Beyond!"
new_text = original_text.swapcase()           ## == "tO INFINITY AND BEYOND!"
new_text = original_text.ljust(30)            ## == "To infinity and beyond!       "
new_text = original_text.rjust(30)            ## == "       To infinity and beyond!"
new_text = original_text.replace(" ", "--")   ## == "To--infinity--and--beyond!"
```

Query content of string

```
text = "To infinity and beyond!"
num = len(text)             ## get length of string ... num == 23
num = text.count(" ")       ## count spaces in string ... num == 3
num = text.index("o")       ## position of first 'o' in the string ... num == 1
num = text.rindex("o")      ## position of last 'o' in the string ... num == 19
result = text.isnumeric()   ## does it contain only numbers?
result = text.isalpha()     ## does it contain only letters?
result = text.islower()     ## is it all lowercase?
result = text.isupper()     ## is it all uppercase?
result = text.istitle()     ## is it all title case?
result = text.isspace()     ## is it all spaces?
```

Example usage:

```
name = "Luke Skywalker"
space = name.index(" ")
given_name = name[:space]
family_name = name[space+1:]
```

Casting

Converting between datatypes is known as casting.

```
f = float( 100 )
i = int( 13.7 )
i = int( "13" )
f = float( "13.7" )
s = str( 13.7 )
```

Initially you will find it most useful to convert the String from `input()` into number types.

```
n = float(input("Please enter a number"))
result = n * 2
print(f"Double your number is: {result}")
```

Problem set

1. For any string that consists of exactly two words with one space separating them, swap the two words around. For example: Given the string `Hello world!`, have the program print `world! Hello`.
2. Given a sentence input, return how many words are in the sentence. For example, `The quick brown fox jumps over the lazy dog.` is `9` words.
3. Given a string input of a date in format, `dd/mm/yyyy`, print an output advising the current day, month and year number.
4. Given a string, return a new string made of 3 copies of the last 2 chars of the original string. Assume the input string length will be at least 2 characters. For example, the string "Hello" should be result in "lololo".
5. Given a string, return the string made of its first two chars, so the String "Hello" yields "He". If the string is shorter than length 2, return whatever there is, so "X" yields "X", and the empty string "" yields the empty string "".
6. Given a string, return a version without the first and last char, so "Hello" yields "ell". The string length will be at least 2.
7. Given 2 strings, return their concatenation, except omit the first char of each. The strings will be at least length 1. For example, strings "Hello" and "There" should result in "ellohere".
8. How would you print the following? `All "good" men should come to the aid of their country.` (ie: how to print the double quote character)
9. Write code that will produce the following printout using only a single print() function call.

```
Hello
Hello again
```

Truthiness

- Video: Learning Python (2018 edition) 02: Making decisions (truth, if, else)

Boolean variables

- A boolean variable has only two possible values, `True` or `False`
- Booleans are important for selective execution of code.
- Assigning boolean variables

```
a = True
b = False
print(a,b)
```

Boolean operations

We can query the values of other variables to determine truth.

```
result = (10 == 12)     # is equal to
result = (10 > 12)      # greater than
result = (10 >= 12)     # greater than or equal to
result = (10 < 12)      # less than
result = (10 <= 12)     # less than or equal to
result = (10 != 12)     # not equal to
```

We can compound multiple queries together into one statement

```
result = (10 > 5) and (10 < 100)
result = (10 > 5) or (10 > 50)
result = not (10 > 5) or (10 > 50)
```

Order of precedence: not, and, or (if in doubt, use parenthesis)

Truth functions for strings

```
string1 = "Some string"
truth = string1.isnumeric()   # does it contain only numbers?
truth = string1.isalpha()               # does it contain only letters?
truth = string1.islower()               # is it all lowercase?
truth = string1.isupper()               # is it all uppercase?
truth = string1.istitle()               # is it all title case?
truth = string1.isspace()               # is it all spaces?
```

You can also query if a sub string is in a larger string...

```
exists = "h" in "hello"
exists = "z" in "hello"
```

For example

```
text = "May the force be with you!"
if "force" in text:
    print("The force is strong with this string")
else:
    print("The force is not with this string")
```

If, elif, else, while

If

Having determined how to calculate truthfulness, we can use that to conditionally execute statements.

```
sister_age = 15
brother_age = 12
sister_is_older = sister_age > brother_age
if sister_is_older:
   print("Sister is older")
```

Rather than storing the truthfulness in a boolean variable, you would normally place the query directly in the `if` statement such as:

```
sister_age = 15
brother_age = 12
if sister_age > brother_age:
   print("Sister is older")
```

- Note the colon and the indentation
- Python will conditionally execute until you return to the previous level of indentation
- This applies whenever a Python line ends with a colon (you'll see it used repeatedly)

Examples of different `if` statements:

```
if 1 == 1:             ## Is 1 equal to 1              ... True
if 1 == 0:             ## Is 1 equal to 0              ... False
if "a" == "a":         ## Is "a" equal to "a"          ... True
if "a" == "A":         ## Is "a" equal to "A"          ... False
if "a" != "z":         ## Is "a" not equal to "z"      ... True
if 1 > 0:              ## Is 1 greater than 0          ... True
if -1 > 0:             ## Is -1 greater than 0         ... False
if 2 >= 3:             ## Is 2 greater or equal to 3   ... False
if -3 < -1:            ## Is -3 less than -1           ... True
if 3 < 1:              ## Is 3 less than 1             ... False
if 2 <= 3:             ## Is 2 less or equal to 3      ... True
```

Else

We can also tell Python to run some alternative code if the comparison was not true. That would look like:

```
sister_age = 15
brother_age = 12
```

```
if sister_age > brother_age:
   print("Sister is older")
else:
   print("Brother is older")
```

The example problem has three scenarios. What will the above do if they are twins (same age)?

Multiple comparisons

We can also join multiple queries together using the `and` or `or` key words such as...

```
a = int(input("Enter a number: "))
if a > 0 and a < 10:
    print("You entered a number between 0 and 10")
if a < 0 or a > 10:
    print("You entered a number less than 0 or greater than 10")
```

Elif

We can link multiple queries together using `elif`. Upon finding the first that resolves to `True`, Python will cease the remaining comparisons.

```
a = int(input("Enter a number: "))
if a > 10:
    print("a is bigger than 10")
elif a > 0:
    print("a is bigger than 0 but not bigger than 10")
elif a == 0:
    print("a is zero")
else:
    print("a is less than 0")
```

Example

A more detailed example combining several elements.

```
name_1 = input("Enter person 1's name:")
name_2 = input("Enter person 2's name:")
age_1 = int(input(f"Enter {name_1}'s age:"))
age_2 = int(input(f"Enter {name_2}'s age:"))
diff = abs(age_1 - age_2)
if age_1 > age_2:
   print(f"{name_1} is {diff} years older than {name_2}")
elif age_1 < age_2:
   print(f"{name_2} is {diff} years older than {name_1}")
else:
   print(f"{name_1} and {name_2} are the same age")
```

Multiple elif's

You can chain as many elif's together as you like.

```
from datetime import datetime
day_of_week = datetime.now().weekday()
if day_of_week == 0:
    print("Today is Monday")
elif day_of_week == 1:
    print("Today is Tuesday")
elif day_of_week == 2:
    print("Today is Wednesday")
elif day_of_week == 3:
    print("Today is Thursday")
elif day_of_week == 4:
    print("Today is Friday")
elif day_of_week == 5:
    print("Today is Saturday")
elif day_of_week == 6:
    print("Today is Sunday")
```

(we'll cover how to use date functionality in more detail later)

Remember:

- Use a double equal sign to compare two values! A single equal is used to **set** the value rather than ask if they are a match.
- End your "question" with a colon and indent the code to run when the comparison is True.
- The `if` statement will keep asking questions of the various `elif` until it finds one that is `True`. After one item is `True`, it will skip the rest of the options available.

While

The `while` loop works very similar to the if statement. Any question you can ask of an `if` statement can be used in a `while` loop. The difference being that so long as something is `True`, it will keep running the same indented section of code. An example:

```
stop_at = int(input("Enter a number for me to count up to: "))
num = 1
while num <= stop_at:
   print( num )
   num = num + 1
print("The end!")
```

Another example...

```
import random
secret = random.randint(1,99)   # generate a random number between 1 and 99
guess = int(input("Guess my secret number between 1 and 99: "))
```

```
while guess != secret:
    if guess > secret:
        guess = int(input("Too high. Guess again: "))
    elif guess < secret:
        guess = int(input("Too low. Guess again: "))
print("Correct!")
```

Problem set - Truth

What range of inputs is required for each of the following to return True

```
a = (int)input("Enter a number:")
truthy = 0 < a and 100 > a
print(truthy)
```

```
a = (int)input("Enter a number:")
truthy = not 0 < a and 100 > a
print(truthy)
```

```
a = (int)input("Enter a number:")
truthy = 0 < a or 100 > a
print(truthy)
```

```
a = (int)input("Enter a number:")
truthy = not 0 < a or not 100 > a
print(truthy)
```

Problem set - Selection & iteration

1. Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included).

2. Suppose you ask the user what the temperature is. Create a program that will respond as follows:

    - If the temperature is between 20 and 27, say that it is "Just right"
    - If the temperature is below 20, say that it is "too cold"
    - If the temperature is above 27, say that it is "too hot"

3. Create a program that allows the user to input the sides of any triangle, and then return True/False to indicate if the triangle is a Pythagorean Triple or not.

4. Write a program which iterates the integers from 1 to 50. For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

5. Write a program to check a triangle is equilateral, isosceles or scalene. An equilateral triangle is a triangle in which all three sides are equal. A scalene triangle is a triangle that has three unequal sides. An isosceles triangle is a triangle with (at least) two equal sides.

6. Write a program to construct the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

7. The fibonacci sequence is created by summing the two previous numbers together. The first 10 numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. Use a `while()` loop to create a program that will calculate the n-th number of the sequence. For instance, if asked for the 8th number, it should provide the answer of 21.

8. Write a program to check the validity of password input by users. The rules for a valid password are:

- At least 1 letter between [a-z] and 1 letter between [A-Z].
- At least 1 number between [0-9].
- At least 1 character from [$#@].
- Minimum length 6 characters.
- Maximum length 16 characters.

9. Write a program that will allow a user to input his name. The prompt and input data would look something like this: `Please enter your name: Peter Ustinov`. Using a for-loop and the String method, `substring()`, produce a printout of the reversal of the name. For example, the name Peter Ustinov would be: `vonitsu retep`. Ensure that the printout is in all lower-case.

10. If we didn't do it as an example together in class (or you are using my notes online), create a simple number guessing game. The program needs to work as follows:

- The computer picks a random number and stores it as a secret number
- Ask the user to guess the number
- If the guess is higher than the secret number, print the message "too high"
- If the guess is lower than the secret number, print the message "too low"
- If the guess is correct, print the message "you are correct!"
- To use a while loop to keep the game going until the correct guess has been made
- Bonus points: Can you keep count of the number of guesses it takes the player to get it correct?

Note: in Python to generate a random number, you should `import random` at the top of your program, and then use an instruction such as `r = random.randint(0, 100)` to generate a random integer between 0 and 99.

For loops ... with a range

You can also use a for-loop when you know the number of iterations you wish to loop in advance.

```
limit = int(input("Enter a number for me to count up to: "))
for number in range(limit):   # will loop from 0 to limit-1
   print( number )
print("The end!")
```

You can also specify a starting number other than zero. For instance

```
for number in range(50, 100):   # will loop from 50 to 99
   print( number )
print("The end!")
```

You can even specify that it counts downwards, or using an interval different to one by specifying a third parameter to the `range()` function.

```
for number in range(100, 0, -1):   # will loop from 100 to 1
   print( number )
print("The end!")
```

Lists

- Video: Learning Python (2018 edition) 03: Lists and loops (lists, for loops)

Why use lists?

A list (known as an array in most other languages) allows us to store a list/set/collection of values all assigned to one variable identifier. They are very useful when we have a collection of values that are similar in nature and that will be processsed in the same manner.

For example, supposed we are keeping record of test scores obtained by a group of students. Without a list we could use something like the following:

```
score1 = 59
score2 = 92
score3 = 85
score4 = 61
score5 = 78
```

Supposed we want to calculate the highesst, lowest and average score? That would look like...

```
highest = score1        # Initially set highest to the first value
if score2 > highest:
    highest = score2
if score3 > highest:
    highest = score3
if score4 > highest:
    highest = score4
if score5 > highest:
    highest = score5
lowest = score1         # Initially set lowest to the first value
if score2 < lowest:
    lowest = score2
if score3 < lowest:
    lowest = score3
if score4 < lowest:
    lowest = score4
if score5 < lowest:
    lowest = score5
average = (score1 + score2 + score3 + score4 + score5) / 5
print(f"The highest score was {highest}, the lowest was {lowest} and the average was {average}")
```

You can see that the whole process will quickly get very tedious. There will be a lot of copy-and-pasting-and-renaming of code going on. Imagine if we needed to scale this up to 100 students for an entire year group? Unmanagable and error prone!

Enter the list!

The equivilant task using lists might look like

```
scores = [59, 92, 85, 61, 78]
highest = scores[0]     # Initially set highest to the first value
lowest = scores[0]      # Initially set lowest to the first value
total = 0               # Running total for calculating the average later
for value in scores:    # Iterate through each `value` within `scores`
    if value > highest:
        highest = value
    if value < lowest:
        lowest = value
    total = total + value
average = total / len(scores)
print(f"The highest score was {highest}, the lowest was {lowest} and the average was {average}")
```

Our scores array can easily contain 1000s of records and we would not have to change a single line of the calculations code! Arrays can be extremely useful!

Creating a lists (or array)

As already said: A list (known as an array in most other languages) allows us to store a list/set/collection of values all assigned to one variable identifier.

Examples

```
primes = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]
vowels = ["A", "E", "I", "O", "U"]
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
empty_list = []
```

Syntax notes:

- The square brackets denotes the beginning and end of the list collection
- Items in a list are separated by commas

Manipulating a list

Adding items

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
starwars.append("BB-8")     # Append to end of the list
starwars.insert(0, "R2-D2") # Insert to position 0
```

Combining lists

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
darkside = ['Vader', 'Palpatine']
characters = starwars + darkside
```

Removing items

```
starwars.remove("Han")      # Remove by item value
starwars.pop(0)             # Remove by item position
```

Overwriting an item

```
conflicted = ['Anakin', 'Kylo']
conflicted[0] = "Darth Vader"   # He turned!
```

Sublists

Lists have many of the same features of strings (which is really just a list of characters) to query them and get sub-parts from.

What is the value of partial after each line?

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
partial = starwars[3:]
partial = starwars[:3]
partial = starwars[-1:]
```

Other list functions

```
numbers = [36, 9, 13, 71, 58, 95, 22]
result = min(numbers)
result = max(numbers)
result = len(numbers)
result = sum(numbers)
numbers.sort()                  # Sorts the current list
numbers.reverse()               # Reverses the currnent list
position = numbers.index(13)    # What position location is 13 at?
occurances = numbers.count(9)   # How many occurances of 9 appear?
is_inside = 13 in numbers       # Does the item 13 appear in list numbers?
```

Splitting and joining lists

Splitting string into a list

```
saying = "May the force be with you"
words = saying.split(" ")
print(words)
```

A more useful example

```
birthdate = input("Your date of birth as dd/mm/yyyy:")
parts = birthdate.split("/")
day = int( parts[0] )
month = int( parts[1] )
year = int( parts[2] )
```

Joining a list into one string

```
delimiter = " "
parts = ["I", "am", "Groot"]
groot = delimiter.join(parts)
print(groot)
```

Quering a list

You can query if an item exists inside a list

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
if "Luke" in starwars:                  ## Is "Luke" in the starwars list?
    print("Luke is in starwars")
else:
    print("Luke is not in starwars")
```

List comprehension

```
# Python's list comprehension is a great little short cut...
vals = [expression for value in collection if condition]

# The long form equivilant is...
vals = []
for value in collection:
    if condition:
        vals.append(expression)
```

For example:

```
nums = [ x*x for x in range(10) if not x % 2 == 0 ]
print(nums)   # [0, 4, 16, 36, 64]
```

Another way of using it is to filter one large list into a smaller list...

```
# Find those numbers divisible by 3...
nums = [0,1,2,3,4,5,6,7,8,9,10,11,12]
threes = [ x for x in nums if x % 3 == 0 ]
print(threes)   # [0, 3, 6, 9, 12]
```

Problem set

```
TODO
```

For loops ... with lists

We can traverse over a list using a 'for-loop'. This is a loop that increments over a series of values, running the intended code with a different value each time.

There are two methods of writing for-loops.

Method 1: For item in collection

The **item in collection** method will, when provided a list of values, the for-loop will iterate through each value. In the following example, the variable `character` shall have the value `Luke` the first time the loop is executed, `Han` the second time, `Leah` the third time and so on until the list is complete.

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
for character in starwars:
    print(f"Vote for {character} as your favourite starwars character")
```

Method 2: For value in range

The **value in range** method will increment of a series of numbers. In the example below, the `range(10)` function generates a sequence of values from `0` up to but not including `10`. So, the print will generate the output of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`.

```
for i in range(10):
    print(i)
```

You can still use this method to work in combination with lists, by using the `len()` function to determine the length of a given list, and then use the square bracket notation to access individual list elements, as the following example illustrates.

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
for i in range( len( starwars ) ):
    print(f"Character #{ i } is { starwars[i] }")
```

Problem set

For some introductory level questions, I recommend solving the problem sets on coding bat:

- List-1 coding.bat problems (no loops)
- List-2 coding.bat problems (require 1 loop)

Be warned, a number of the questions in the main problem set are quite challenging for new programmers. Do not worry if you feel some of them are beyond you if you are at the beginning stags of learning to program. Revisit the questions you can't do after you have been programming consistently for about 12 months.

1. Write a program to sum all the items in a list.
2. Write a program to get the largest number from a list.
3. Write a program to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings.
4. Write a program to remove duplicates from a list.
5. Write a function that takes two lists and returns True if they have at least one common member.
6. Write a program to print a specified list after removing the 0th, 4th and 5th elements. Sample List : `['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']` Expected Output : `['Green', 'White', 'Black']`
7. Write a program to print the numbers of a specified list after removing even numbers from it.
8. Write a program to select an item randomly from a list, which is then removed from the original list so it can’t be re-drawn (just like a deck of cards scenario)
9. Write a program to generate and print a list of first and last 5 elements where the values are square of numbers between 1 and 30 (both included).
10. Given two lists, write a program to print the items that are not in both lists.
11. Write a program to append the items from one list to a second list.
12. Write a program for computing primes upto 1000. Hint: Google for the Sieve of Eratosthenes

For question 8, you may like to copy and paste the following as a list to use:

```
deck = [
    "A♥", "2♥", "3♥", "4♥", "5♥", "6♥", "7♥", "8♥", "9♥", "10♥", "J♥", "Q♥", "K♥",
    "A♠", "2♠", "3♠", "4♠", "5♠", "6♠", "7♠", "8♠", "9♠", "10♠", "J♠", "Q♠", "K♠",
    "A♦", "2♦", "3♦", "4♦", "5♦", "6♦", "7♦", "8♦", "9♦", "10♦", "J♦", "Q♦", "K♦",
    "A♣", "2♣", "3♣", "4♣", "5♣", "6♣", "7♣", "8♣", "9♣", "10♣", "J♣", "Q♣", "K♣",
]
```

Functions

- Suggested video Python Functions by Socratica

Introducing functions

Functions are blocks of code that you assign a name to. You can use that name to easily run that code again whenever you need.

Functions are very useful for separating common tasks out from your main code. It allows you to avoid repeating yourself all the time which makes your code easier to maintain. Tasks like reading from a file, saving to a file, etc are all ideally suited to being chopped off into a separate function.

Think of an Icecreamary... Lots of different possible flavours, toppings, numbers of scoops, choice of waffle or regular cone, etc.

One person could order a double scoop of chocolate fudge and vanilla on a waffle cone, where as the next customer might ask for a cup of raspberry sorbet with nut sprinkles. The salesperson calculates the cost for each and advises each custoemr on the price. In order to calculate that cost there are a number of inputs (number of scoops, type of cone, etc) and an output (price). How it is actually calculated is not important, provided it is trustworthy and works reliably.

In this way a function can provide a "black box" model through which we can create an abstraction to represent our problem.

Programmers need to know how to (a) use other peoples abstractions and (b) be able to create their own. For now, the abstraction we are concerned with is creating a function.

A couple of examples:

```
def area_of_circle( radius ):
    PI = 3.1415
    a = PI * radius ** 2
    return a

def circumference( radius ):
    PI = 3.1415
    circ = 2 * PI * radius
    return circ

def area_of_triangle( base, height ):
    area = 0.5 * base * height
    return area # send this value back to the code that ran the function

# Execute our custom functions
print( area_of_circle( 10 ))            # prints 314.0
print( area_of_circle( 15 ))            # prints 706...
print( circumference( 5 ))              # prints 31.4
print( circumference( 100 ))            # prints 628.2...
print( area_of_triangle( 10.0, 15.0 ))  # prints 75.0
```

```
print( area_of_triangle( 7.0, 12.0 ))   # prints 42.0
print( area_of_triangle( 4.0, 5.5 ))    # prints 11.0
```

You can see that rather than having to re-write the same process multiple times, we have defined the function once and then been able to reuse it several times with different values as required.

Syntax explainer

Functions:

- The `def` keyword denotes you are defining a function.
- You must have the parenthesis after the function name. If no parameters are required, just them empty.
- Like previous uses, indentation is used to indicate which code belongs to the function. Return to the previous level of indentation when the function is complete.
- The return at the end of the function is optional. If there is no result to pass back to the code that called it, you can skip it.

Functions in seperate files

Putting functions you will frequently reuse in different files makes it easy to import them into other projects later.

File: circles.py

```
def area(radius):
    PI = 3.1415
    a = PI * radius ** 2
    return a

def circumference(radius):
    PI = 3.1415
    circ = 2 * PI * radius
    return circ
```

File: main.py

```
# Import the circles file
import circles

# Prompt the user
r = int(input("What is the radius of your circle?"))

# Execute our functions
answer1 = circles.area(r)
answer2 = circles.circumference(r)

# Output the results
print(f"The area of your circle is: {answer1}")
print(f"The circumference of your circle is: {answer2}")
```

Default and optional parameter values

```
def greetings( given_name, family_name=None ):
    # Providing a family_name is optional in this example
    if family_name:
        print(f"Hello {given_name} {family_name}")
    else:
        print(f"Hello {given_name} Doe")

greetings("Jane", "Smith")
greetings("John")
```

Functions for user input validation

Functions can be a handy way to require the user to comply with our wishes to enter information in a particular manner. By the time we write the checking/validation code and the loop, user input checks can run to several lines, and it would be quite common within a simple program to want to validate the same style of input several times. Functions make a handy way to reuse code for this purpose.

```
def confirm( prompt ):
    loop = True
    response = ""
    while loop:
        response = input( prompt )
        if response == "y" or response == "n":
            loop = False
        else:
            print("Only a 'y' or 'n' character are accepted, please try again.")
    return response
```

Problem set

1. Create a function `area_right_angled_triangle(base, height)` that returns the calculated area.
2. Create a function `area_non_right_angled_triangle(base, height, angle)` that returns the calculated area (remember you will need to convert the angle to radios before using it with the sine function).
3. Create a user input validation function that requires the input of a number. Hint: Remind yourself how the `.isnumeric()` function of strings works.
4. Create a user input validation function that requires the input of a phone number (so `+`, spaces and `-` characters are permitted). Hint: Remind yourself how checking for `if 'z' in "String":` works.
5. Create a user input validation function that requires the input of a date in the `dd/mm/yyyy` format. Bonus points if you ensure that the `dd`, `mm` and `yyyy` values make sense (ie: day should be between 1 and 31). Hint: use `.split()`.
6. Create a user input validation function that accepts a list of strings as the parameter and presents them to the user as a list of menu choices, requiring the user to enter a number corresponding to a valid choice before proceeding. For example if the code to run the funciton was....

```
menu = ["Open file", "Save file", "Quit program"]
choice = menu_picker( menu )
print( f"You choose option {choice}" )
```

The output could look like...

```
Your choices are:
1. Open file
2. Save file
3. Quit program
Please enter a number from 1 to 3:
```

Files

- Suggested video Text files in Python by Socratica

Read entire file as a string

```
with open("countries.txt", "r") as f:   # Open file for reading
    content = f.read()                  # Load entire file into 1 large string
    print(content)                      # Do something with the string
```

Read entire file as a list, one string per line

```
with open("countries.txt", "r") as f:   # Open file for reading
    content = f.read()                  # Load entire file into 1 large string
    lines = content.splitlines()        # Split into lines
    for line in lines:
        print(line)
```

Writing a text file - Using just a string

```
content = "My exciting material"
with open("stuff.txt", "w") as f:   # Open file stuff.txt for writing
    f.write(content)                # Write this string to the file
```

Writing a text file - Using a list of strings

```
content = ['Leah', 'Obi-wan', 'Yoda', 'Rey', 'Finn', 'bb-8']
save = "\n".join(content)           # Convert list to a string, adding a new-line character after
each string
with open("people.txt", "w") as f:  # Open file people.txt for writing
    f.write(save)                   # Write this string to the file
```

Add to a file without replacing the original content

```
content = "More exciting material"
with open("stuff.txt", "a") as f:   # Open file stuff.txt for appending
    f.write(content)                # Add this string to the file
```

File opening modes:

- `r` for reading
- `w` for writing (erasing it if it exist)
- `a` for appending (add to file without erasing previous content)

Remember

- The `with` statement will close the file when you unindent
- `.splitlines()` behaves like `.split("\n")`
- For greater predictability, cast everything to strings before writing to files

About files...

- The `with` statement will close the file when you unindent
- `.splitlines()` behaves like `.split("\n")`
- Opening a file using `w` mode will overwrite an existing file
- For greater predictability, cast everything to strings before writing to files

By the way, just before we finish the section on reading/writing files, you may have wondered what the "r" or "w" in the `open()` function meant. This instructs Python how we want to access the file we request. The different modes for opening a file that will be relevant for now are as follows:

- `r` for reading
- `w` for writing (erasing it if it exist)
- `a` opens for appending

OS tools

There is a range of operating system functionality that you will commonly use with working with files.

Some functions that will be useful

```
# Import the Operating System module
import os

# Check if a file or folder exists of a given name
if os.path.exists("filename.txt"):
    print("The item exists")

# Check if a file exists
if os.path.isfile("filename.txt"):
    print("The item is a file")

# Check if a folder exists
if os.path.isdir("documents"):
    print("The item is a folder/directory")

# Delete a file
# - USE WITH CAUTION
os.remove("file.txt")

# Rename a file
os.rename("oldfile.txt", "newfile.txt")

# Create a folder
# - Will error if it already exists
os.mkdir("folder")
```

```
# Remove a folder
# - Will error if it is not empty
os.rmdir("folder")

# Not relevant to files, but fun to know about....
# Get current logged in user
username = os.getlogin()
```

Problem set

1. Read a text file into a string, print it to the screen.
2. Read a text file into a list of strings, and print out the number of lines in the file.
3. Ask the user for the name of a file they'd like to create. Ask the user to type an input, and then save that as the content of the file.
4. Ask the user for the name of a file they'd like to create. Using a while loop, keep ask the user to type an input and only stop when they enter an empty input. Save all the lines entered as the content of the file.
5. Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to read. Print just the content of that line to the user.
6. Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to change. Prompt the user for the new content of the relevant line. Write to the file the new list of strings.

Challenge question

Write a simple to-do app. Each line represents an task. The first character should be `-` if the task still needs doing, and `x` once it has been marked as complete. When the program starts, it should present the user with 3 options as follows.

```
Welcome to simple-to-dos!

* Enter the task number to mark it as complete,
* Hit enter on an empty line to quit, or
* write some text to add a new task.

The tasks currently pending are:

1 Cook dinner
2 Clean bedroom

Your input >>
```

An example of the text file follows.

```
x Write blog post
- Cook dinner
- Clean bedroom
x Make weekly tiktok
x Subscribe to Mr B TV
```

Exceptions

- Suggested video Exceptions in Python by Socratica

Study the following code and ask, what will Python do if the user input's `0`? Test your hypothesis.

```
denominator = float(input("Please enter a number: "))
result = 100.0 // denominator
print(f"100 divided by {denominator} is {result}")
```

Exceptions are the error events that occur when your program is running that Python can not recover from on its own, so will result you your program crashing out.

We can avoid program crashes from exceptions. If we are creating code that we consider may result in an exception, we can preempt it in our code providing Python with an alternative to take. This may be referred to as catching the exception.

Catching an exception

Firstly, we can have a generic "catch every exception" response with a "try and except" set of blocks.

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denominator
    print(f"100 divided by {denominator} is {result}")
except:
    print("I can't do that!")
```

A warning

You should be aware that while possible, the failsafe catch-all of `except:` is considered poor practice. This is because it hides bugs in your code since any error in your code will create an exception.

The following excample will illustrate the problem...

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denoninator
    print(f"100 divided by {denominator} is {result}")
except:
    print("I can't do that!")
```

This code looks the same as before but has one critical different. I have inserted a deliberate bug... intentionally made hard to find... can you spot it? Hint: There is a spelling error on a variable

name.

This bug means no matter what the user enters when they run the program, it will fail and produce the `I can't do that!` message. This would become very frustrating for you as the programmer as you won't understand why your code is not working.

Catching specific exceptions

A better solution is to specify which errors we are anticpating and to catch those specifically. To find the label to use for the exception statement, you can either run the code in such a way as to generate the error (and thus read it from the error statement), or check the Python Exceptions documentation at the link for the description of each official exception.

- https://docs.python.org/3/library/exceptions.html

A better solution to our problem is thus...

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denominator
    print(f"100 divided by {denominator} is {result}")
except ValueError:
    print("That wasn't a number")
except ZeroDivisionError:
    print("I can't divide by zero")
```

Raising exceptions

Occassionally you might have a valid cause to want to Python to have an exception event.

We can use the `raise` statement to do this.

Raising a generic exception like shown here however is considered bad practice and should be avoided. For the same reason as catching generic exceptions... they hide bugs.

I generally only ever use this approach when I am developing a program and want to quickly test if a particular section works as intended or not, and so would want the program to fail. Deliberately causing your program to fail when in real-world use is very bad practice and lazy programming. You should program your way around it with code that responds to errors rather than causing them.

```
print("I am about to fail")
x = 10
if x > 5:
    raise Exception("Not allowed to have a number greater than 5")
print("You will never see me print")
```

Raising exceptions is all beyond the scope of your course. If you are interested in more on this, I suggest getting the relevant Python documentation at..

- https://docs.python.org/3/tutorial/errors.html#raising-exceptions

Problem set

Do not use the generic exception handler in your responses to these problems.

Q1. Create a quadratic formula calculator that uses exceptions...

- to ensure the `a`, `b` and `c` entered by the user are numbers
- to detect if `a` is zero (would result in a division by zero)
- to detect if there are no real solutions (square root of a negative number exception)

Q2. Question 5 of the Files problem set reads *"Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to read. Print just the content of that line to the user"*.

- Add exception handling in case the file does not exist.
- Add exception handling in case the user asks for a line number beyond the limits of the list (ie: if there are only 5 lines and the user askes for line 6).

Dates & times

Creating a datetime

Python uses the `DateTime` object to provide a programmer-friendly interface for managing all the different aspects of dates and times. This includes keeping track of leap years, the number of days in each month, timezones, day light saving where applicable and many other complications that start to appear where ever dates and times are involved.

```
from datetime import datetime

# Create a datetime using current computer date & time
now = datetime.now()

# Create a datetime with year=2019, month=12, day=25
christmas = datetime( 2019, 12, 25 )
print(christmas)

# Create a datetime with year=2019, month=12, day=25, hour=11, minute=00, seconds=00
christmas = datetime( 2019, 12, 25, 11, 00, 00 )
print(christmas)

# Create a datetime from a formatted string
# - See section below about formatting the string
birth_text = input("What is your birthday (write it as dd/mm/yyyy) ?")
birth_date = datetime.strptime(birth_text, "%d/%m/%Y")
print(birth_date)
```

Using timestamps

A timestamp is how computers internally store date and time information. Historically this is internally stored as the number of seconds since the computing epoch, deemed as 01/01/1970 00:00 UTC.

```
from datetime import datetime

# Create a timestamp based on current date/time
timestamp = datetime.now().timestamp()

# Create a datetime from a timestamp
timestamp = 1563958625      # Number of seconds since 01/01/1970 00:00 UTC
july24_2019 = datetime.fromtimestamp(timestamp)
print(july24_2019)
```

Differences between dates

We use the `timedelta` object to perform additions or subtractions with dates and times.

- `timedelta` accept sany combination of the following options: days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0

```
from datetime import datetime, timedelta

apollo_11 = datetime( 1969, 7, 20, 20, 17, 40 )
now = datetime.now()

# Create a timedelta automatically by subtracting two dates
diff = now - apollo_11
print(f"Apollo 11 landed {diff.days} days ago!")

# Create a new date by adding a timedelta to a date
ten_thousand = now + timedelta( days=10000 )
print(f"10'000 days from today is { tenthousand.strftime("%d %B, %Y") }")
```

Create formatted strings

While the `datetime` object is useful from a programming perspective, it will usually need converting to/from strings when being presented to the user, or receiving from the user.

Use `strftime()` to generate strings containing dates and times in human presentable formats for your users.

```
# Create a date
apollo11 = datetime( 1969, 7, 20, 20, 17, 40 )

# Make pretty, human readable versions
pretty_date = apollo11.strftime("%A, %d %B, %Y")
pretty_time = apollo11.strftime("%H:%M:%S")

# Do something with them
print(f"Apollo 11 landed on the moon on {pretty_date} at the time of {pretty_time}")
```

Date based codes

- %a - Weekday abbreviated (eg: Sun)
- %A - Weekday full name (eg: Sunday)
- %d - Day number in month (zero padded eg: 02)
- %b - Month name abbreviated (eg: Jan)
- %B - Month full name (eg: January)
- %m - Month number (zero padded eg: 01)
- %y - Year without century (zero padded)
- %Y - Year with century (zero padded)

Time based codes

- %I - Hour 12 hour clock (zero padded)
- %H - Hour 24 hour clock (zero padded)

- %M - Minute (zero padded)
- %S - Second (zero padded)
- %p - AM or PM

Get parts of dates/times

To retrieve parts of a date or time

```
from datetime import datetime

apollo_11 = datetime( 1969, 7, 20, 20, 17, 40 )
y = apollo_11.year        # 1969
m = apollo_11.month       # 7 (July)
d = apollo_11.day         # 20
hr = apollo_11.hour       # 20 (8:00pm in 24 hr time)
mi = apollo_11.minute     # 17
se = apollo_11.second     # 40
wkd = apollo_11.weekday() # 6 (0=Monday so 6 is Sunday)
```

Replace parts of a date

Use the date `.replace()` function

Example 1

```
from datetime import datetime
date_1 = datetime(1980, 6, 20)
date_2 = date_1.replace( year = 2019 )  # Replace the year
print(date_2)   # 20/06/2019
```

Example 2

```
from datetime import datetime

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
typed = input("What is your birthday (write it as dd/mm/yyyy) ?")
birthday = datetime.strptime(typed, "%d/%m/%Y")
now = datetime.now()
y = birthday.replace( year = now.year )  # Replace the year
day_number = y.weekday()
print(f"Your birthday this year is a {days[ day_number ]")
```

Problem set

1. Write a function that, given a string in date format, will calculate and return your age in years. Example `getAge("01/01/2010") returns 9`.
2. Write a function that, given a string in date format, will calculate and return the number of days until the date. `get_days_until("01/01/2021") returns 312`.
3. Write a function that, given a string in date format, will calculate and return the day of week as a string for that date. `get_day_of_week("01/01/2010") returns "Tuesday"`.
4. Write a function accepting two dates that will return the number of days between the two dates. Example function call being `get_days_between( "04/06/2018", "02/08/2016" )`

Dictionaries

- Suggested video Python Dictionaries by Socratica

Python dictionaries are similar to lists. Where a list uses a number to keep track of the individual elements contained within it, a dictionary (generally) uses a word.

Create an empty dictionary

```
person = { }    # Curly braces instead of the square brackets used for lists
```

Set values to your dictionary

```
# Create an empty dictionary
person = { }    # Curly braces instead of the square brackets used for lists

# Set values to your dictionary
person["given_name"] = "Paul"
person["family_name"] = "Baumgarten"

# Get elements from the dictionary
print( person["given_ame"] )
print( person["family_name"] )

# Add / modify elements in the dictionary
person["email"] = "pbaumgarten@isl.ch"
person["website"] = "https://pbaumgarten.com"

# Remove an element from the dictionary
del person["website"]
```

Loop through all the elements of the dictionary

```
for key,val in person.items():
   print(f"field {key} has value {val}")
```

JSON

A JSON file is a convienent file format for storing dictionary data. It is derived from the world of Javascript.

The following are two example functions that can be used for reading and writing JSON files.

Convert a dictionary/list structure into a JSON string (useful for saving to a file)

```
import json

# Convert to JSON text string
json_text = json.dumps( person )

# Save it to a file
with open("person.txt", "w") as f:
    f.write( json_text )
```

Convert a JSON string into a dictionary/list structure (usefil for loading from a file)

```
import json

# Load from a file
with open("person.txt", "r") as f:
    content = f.read()

# Convert string text to dictionary/list structure
person = json.loads( content )
```

Problem set

> You better believe these are coming soon! 😃

- Dictionary questions
- JSON file read/write questions
- Requests questions