SoloLearn Python 3.x

Basic Concepts

- High level language
- Compiled at runtime
- CPython implementation is used

www.python.org for free download
- Includes IDLE Python GUI (Graphical User Interface)

print() - prints something to the console

" " -or- ' ' denotes a string
ex. print("Hello World!")

quit() - quits out of the console

Ctrl-Z - quits out of the console
Ctrl-C - stops the current program

+, -, /, * - standard arithmetic operators
- follows PEMDAS
- / creates a float number (with decimal)
- negative numbers are permitted
- dividing by zero will give an error

Floats - used to represent a number that is not an interger
- intergers operated on with floats are converted to floats

** - signifies exponent
// - floor division (no remainder)
% - modulo (only remainder)

String - text that is used or stored
- denoted by " " -or- ' ' (no difference)

Escape - overiding a charachters normal use
- done with \ before the charachter
  ex: 'Brian's' → Brian
      'Brian\'s' → Brian's
- can also be used for new line (\n), tab (\t), and other unicode

input() - asks the user for input and saves it as a string

Concatenation - "adding" strings to make one combined string
- done by putting + between strings
  ex. "spam" + "eggs" = "spameggs"
- strings and numbers cannot be added
- strings can be multiplied by intergers (not floats)
  ex. "spam" * 3 = "spamspamspam"

int() - converts something to an interger
float() - converts something to a float
string() - converts something to a string
- can be applied with input to take in a number
  ex. int(input("Enter a number: "))

Variable - storing a value by assigning it a name
- denoted by name = value
- can swap types depending on what it is currently storing
- letters (case sensitve), numbers (not starting), underscores only
- can't reference variables that are not assigned

In-Place Operator - shorthand that does something else

ex. $x = x + 3 \iff x += 3$ (same for $-, *, /, \%$)

- Can be used on integer, floats, and strings

- Python files end in .py

Control Structures

Boolean - data type that can only be True (T) or False (F)

- Can also be created by comparing values (including strings)

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal

- Spacing (especially indentation) is very important in python as it deliniates code blocks through indentation

if - evaluates code block if expression is true

```
if expression:
    code
    block
end of code block
```

- other control statements can be put inside control statement code blocks
  - referred to as nested

else - comes after an if and evaluates code block if preceding if expression was false

```
if expression:
    code
    block
else:
    code
    block
```
end of if/else

elif - used to nest if/else statements

```
if expression:
    code
    block
elif expression:
    code
    block
else:
    code
    block
```

and - evaluates to T iff (if and only if) both arguments are T

or - evaluates to T if one or both of its 2 arguements are T

not - inverts one arguement (T → F, F → T)

Operator Precedence - order they are resolved in normally

First (**) → (~) → (*, /, %, //) → (+, -) → (>>, <<) → (&) →

→ (^, |) → (<=, >=, <, >) → (<>, ==, !=) →

→ (=, %=, /=, //=, -=, +=, *=, **=) → (is, is not) →

→ (in, not in) → (not, or, and) Last

while - executes a code block continously while a statement is T

```
while expression:
    code
    block      } iteration
end of while
```

- infinite loop if the expression always equates to T

break - causes whatever loop it is apart of to break instantly, moving to the next part of code
  - error if used outside of a loop

continue - causes whatever loop it is apart of to go back to the beginning without running the rest of the iteration
  - error if used outside of a loop

list - object that stores an indexed list of items
  - creation: list_name = [thing_0, thing_1, ..., thing_n] ← length=n+1
  - access: list_name[n]
    n - index of item you want (starting with 0)
  - empty: empty_list = [] ← length=0
  - can hold mixed objects, even other lists

```
things = [0, [1, 2], 3]
things[1][1] == 2
list 1 index ↑   ↑ list 2 index
```

- Strings can be indexed like lists where each charachter takes 1 spot

- objects in the list can be reassigned
- lists can be added and multiplied as strings

in - checks to see if an item is contained within a list
- checks to see if a string is found within another string

object in list
-or-
string in string
} returns T or F

- can be combined with not to see if it is not contained

object not in list
string not in string
} returns T or F

.append() - method of list class
- adds item to end of a list

list.append(item)

len() - returns the number of items in a list

.insert() - method of list class
- adds item to specefic index in the list

list.insert(index, item)

.index() - method of list class
- returns index of first occurence of desired item
- error if the list does not contain the item

list.index(item)

max() - returns largest item in a list
max(list)

min() - returns smallest item in a list
min(list)

.count() - method of list class
- returns how many times an object appears in a list
list.count(object)

.remove() - method of list class
- removes an object from a list
list.remove(object)

.reverse() - method of list class
- reverses the order of a list
list.reverse()

range() - creates a range object from a to b (not including) by step
↑ must be integer
range(a, b, step)
ex. list(range(0,10,2)) == [0, 2, 4, 6, 8]

for - type of loop that goes through an object with an iteration each time
- behaves much like foreach in other languages

```
for item in object:
    code
    block
after for loop ends
```
} great for going through lists

```
for counter in range(n):
    code
    block
after for loop ends
```
} great for iterating code block n times

Functions and Modules

. Keep code DRY (Don't Repeat Yourself) not WET (Write Everything Twice)
  - done with loops, functions, modules

Functions - do things with arguments (args)

```
function_name(arg1, arg2, ..., argn)
```

def - defines a new function (must be defined before call)

```
def f_name(arg1, arg2, ..., argn):   (args do not exist outside of function)
    function
    code
end of function
```

return - used within a function to return a value when it is called
- cannot be used outside of a function
- operates like a break as it will leave the function without running the rest of the code

Comment - inserted to make code readable
- inserted with # (octothorpe or hash)
- ignores all text on line after it is entered

docstring - like a comment but explain code in a broader sense
- inserted with """ """ always after function definition but before first line

```
def f_name():
    """
    docstring
    """
    function
    code
end of function
```

- functions can be renamed or called through multiple names like variables
- functions can be used as args to other functions

Module - code written that can then be imported into other code so that it can be used

- import module_name, brings in a module

ex. import random

random.randint(1,6)

(random: module; randint: variable in module; 1, 6: args)

- from module_name import variable, brings in one variable from the module

★ functions can be variables ★

- from module_name import variable as var_name, imports one variable from module and renames it

Standard Library - all modules pre-installed with Python

- complete documentation on www.python.org

Third Party - modules written by others

- installed with pip (installed by default)
- go to command line/prompt, pip install library_name

Exceptions and Files

Exception - when something goes wrong and the code stops
- all different types of errors depending on what happened

- Exceptions can be expected / handled using try/except

try - tries a code block

except - code block that runs if the specefied exception happened in the try block

```
try:
    code
    block
except exception:
    code
    block
```

- can have multiple excepts per try
- except can handle more than 1 exception

```
except (exception1, exception2, ...):
```

- except with no exception stated will catch all exceptions

finally - used with try/except to run a code block no matter what

```
finally:
    code
    block
```

- Code in finally runs even if an uncaught exception occurs

raise - raises an exception even if none occur
- breaks the code unless it is handled

```
raise exception_name
```

- exceptions can be raised with args for more detail

```
raise exception_name("Description")
```

- raise can be used in an except block to raise caught error

```
except:
    raise
```

Assertion - checks a boolean statement and throws an exception if false

```
assert statement
```

- can take a second argument as description

```
assert statement, "Description"
```

open() - opens a file

```
open("filename") # if file is in same directory
open("path")     # if file is not in same directory
```

- can open file 4 different ways

```
open(file, "w") # write mode (deletes content)
open(file, "r") # read mode (default)
open(file, "a") # append mode (add to end)
open(file, "mode"+"b") # opens in mode + binary for
                       non-text files
```

.close() - method of file object

- closes the file so it can no longer be used

.read() - method of file object
- reads the file

```
file.read()  # reads the whole file
file.read(n)  # reads n bytes of the file
```

- returns an empty string once completely read

.readlines() - method of file object
- returns a list where each item is one line

```
file.readlines()
```

- can also use a for loop to iterate through lines

```
for line in file:
    print(line)
```

.write() - method of file object
- writes argument to the file

```
file.write("text") # writes text to file
```

★ file must be opened with "w" ★

- returns number of bytes successfully written

- make sure files are closed after being opened

```
try:
    file = open("filename.txt")
finally:
    file.close()
```

-or-

```
with open("filename.txt") as name:
    name.read()
```

Both ensure the file is closed

More Types

None - object used to represent nothing
- similiar to null in other languages
- returned when a function doesn't return anything else

```
a = None
```

Dictionary - data structures to map keys to values

```
dict = {"Key1":value1, "Key2":value2, ...} #creates dict

dict["Key1"] # brings up the key's values
```

- values can be any kind of data
- Key Error if unknown key is called

```
dict["key"] = value # reassigns or creates a new key
```

- can use in/not in to check for existance of keys

.get() - dictionary method
- returns value if key is found, None or custom value if not found

```
dict.get(key) # no specefied return
dict.get(key, "error") # specefied return
```

Tuples - immutable (cannot be changed) lists

```
tuple = (val1, val2, ...)
```

-or-

```
tuple = val1, val2, ...
```

. lists can be sliced to select a range of values

```
sublist = list[start:stop] # start index included but stop is not
```

```
sublist = list[start:] # start index to end of list
sublist = list[:stop] # beginning of list to (not including) stop
sublist = list[start:stop:step] # start to stop by step
```

- can use negatives as well to count from end or step backwards

- Lists can be built dynamically

```
cubes = [i**3 for i in range(5)]
evens = [i**2 for i in range(10) if i**2 % 2 == 0]
```

.format() - string object method
- used to substitute arguements into placeholders of strings

```
"{0} {1} {2}".format(0, 1, 2) == "0 1 2"
"{x}, {y}".format(x=5, y=12) == "5, 12"
```

.join() - string object method
- joins a list of strings with another

```
", ".join(["1", "2", "3"]) == "1, 2, 3"
```

.replace() - string object method
- replaces one substring in a string with another

```
"Hello ME".replace("ME", "World") == "Hello World"
```

.startswith() - string object method
- determines (boolean) if string starts with a substring

.endswith() - string object method
- determines (boolean) if string ends with a substring

.upper() - string object method
- changes string to all upper case

.lower() - string object method
- changes string to all lower case

.split() - string object method
- splits string into list by seperator

"1, 2, 3".split(", ") == ["1", "2", "3"]

min() - returns minimum number from a list
max() - returns maximum number from a list
abs() - returns absolute value
round() - rounds a number to a certain # decimal places
sum() - sums a list of numbers

all - goes through list returns T if all T (and)
any - goes through list returns T if any T (or)

enumerate - goes through index and value simultaneously

```
if all([i>5 for i in nums]):
    (any)
for v in enumerate(nums):
```

Functional Programming

- Programming is based around higher-order functions
  - functions that take in or return other functions

Pure - function that depends only on its arguments
Impure - function that changes something else's state

- Pure functions can complicate I/O (Input/Output) and be difficult to write but are better long-term

Lambda - one shot simple function (limited to one expression)

```
(lambda var: expression)(call)
```

- can be assigned to variables and used as functions

```
f = lambda x: x*2
f(2) == 4
```

Map - evaluations a function over each item in an iterable

```
new_iterable = list(map(func, iterable))
```

Filter - removes items from an iterable by a boolean function

```
filtered = list(filter(func, iterable))
```

Generator - iterable that doesn't allow indexing except through for loop

- created with functions and yield
- function behaves like the iterable

```
def infinite_sevens():
    while True:
        yield 7

for i in infinite_sevens:
    print(i)
```

- can also be used to create lists

```
gen_list = list(infinite_sevens)
```

- generators have lower memory usage

Decorators - modifies functions through functions
- used to add to an already completed function
- usually have a wrap function inside decorator function
- more detail needed

Recursion - functions that call themselves
- allows problems to be broken up easier
- Base case - end of the recursion (exit condition)

```
def factorial(x):
    if x==1:  # base case
        return 1
    else:
        return x * factorial(x-1)
```

- without a base case it will run forever
- needs a lot of practice

Sets - similior to lists

```
ex_set = set([1, 2, 3])  # creation
```

- cannot contain duplicate elements
- is not indexed
- .add() adds to a set
- | (union) combines sets
- & (intersection) combines sets with elements only in both
- - (difference) items in the first set but not the other
- ^ (symmetic difference) items in either set but not both

itertools - standar library for iteration

- count(n) - counts up indefinitely from n
- cycle(list) - infinitely iterates through an iterable
- repeat(object) - repeats an object indefinetely or set number times
- takewhile(function, iterable) - removes items from iterable while the function remains true
- chain(iterable, ...) - combines iterables
- accumulate(iterable) - returns running total of values
- product(iterable, ...) - cartesian product of iterables
- permutation(iterable) - returns list of permutations of elements

Object Oriented Programming

Classes - blueprint for an object

```
class Cat:                                   # always first; attributes
    def __init__(self, color, legs):        # most important, class constructor
        self.color = color                   # setting initial values
        self.legs = legs
    def bark(self):                          # method of cat class
        print("woof")

felix = Cat("ginger", 4)                     # init called with class name; creating new object of Cat class
felix.bark()                                 # calling method of Cat class
```

- can declare class attribute true of all objects of that class before init

Inheritance - share functionality between classes
- creating a class that all inheriting classes are

```
class Animal:          ← superclass
    def __init__(self, name, color):
        self.name = name
        self.color = color
                  ↙ inherits from animal class
class Dog (Animal):    ← subclass
```

- methods / attributes of same name in subclass will overwrite superclass
- super() will refer to the superclass

Magic Methods - methods with 2 underscores at beginning and end
- used for operator overloading among things
  └ defining operators on classes that do new things

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)
```

__add__	+	__floordiv__	//	__xor__	^	__eq__	==
__sub__	-	__mod__	%	__or__	|	__ne__	!=
__mul__	*	__pow__	**	__lt__	<	__gt__	>
__truediv__	/	__and__	&	__le__	<=	__ge__	>=

ex. x+y → x.__add__(y) AND OTHERS

- Objects are created, manipulated, then destroyed
  - destruction happens automatically based on reference count
- encapsulation - packaging things into objects
- data hiding - not letting the user mess things up

  _object - weakly private
  __object - strongly private
  } can still be accessed

Class Methods - methods called by a class (cls)
- used to an object using different parameters
- marked with @classmethod decorator

```
class Rectangle:
    def __init__(self, width, height):
        ⋮

    @classmethod
    def new_square(cls, side_length):   # cls: class
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
```

Static Methods - like class methods but recieve no additional args
- marked with static method decorator (no self or cls)

```
class Pizza:
    ⋮
    @staticmethod
    def validate(toppings):
        ⋮

Pizza.validate(ingredient)
pizza = Pizza(ingredient)
```

Properties - customizing access to instance attributes

- put property decorator above a method
- can make attribute read-only

```
@property
def pinapple_allowed(self):
    return False
```

- can be changed with setter/getter function

```
@pineapple_allowed.setter  ← decorator same for getter function
def pineapple_allowed(self, value):
    if value:
        password = ...
        if password == ...
            self._pineapple_allowed = value
        else:
            raise ValueError("Intruder")
```

Regular Expressions

regular expression - string manipulation tool

- domain specefic language (DSL)
  - highly specefic minilanguages
  - eg. regular expressions, SQL

```
import re
```

- usually used with raw strings (no escapes)

```
string = r"string"
```

re.__

.match(pattern, text) - returns boolean if text matches the pattern

- .search(pattern, text) - finds matches for the pattern anywhere in the text
- .findall(pattern, text) - finds all pattern in text and returns them as a list

- search returns an object
  - .group() - returns matched string
  - .start() - index of first charachter of first match
  - .end() - index of last charachter of first match
  - .span() - tuple of first and last charachter index of first match

- .sub(pattern, replace, string, max) - replaces pattern in string with replace max # of times
  - returns the modified string

metacharachters - charachters that have new meanings in raw stings

. - matches any other charachter other than new line

pattern = r"gr.y" matches grey and gray

^ - matches start of a string

$ - matches end of a string

pattern = r"^gr.y$" matches to strings starting with gr, then any charachter, then ending with y

character class - way to match specefic charachters

pattern = r"[aeiou]"

search will match texts conting any of those charachters

- can also work with ranges

pattern = r"[A-Z][a-z][0-9]"

will match to string with an upper case → lower case → number

^ - inverts the range to include anything but

.$,. have no meaning

pattern = r"[^A-Z]"

will match anything but upper case

* - looks for zero or more repetitions

pattern = r"[egg(spam)*]"

will match egg followed by 0 or more spam

+ - looks for one or more repetitions

pattern = r"g+"

will match one or more g

? - looks for zero or one matches

pattern = r"ice(-)?cream"

will match to ice → 0 or 1 - → cream

{x,y} - looks for between x and y repetitions

pattern = r"9{1,3}$"

will match to 1-3 9

{x,} - x to infinity

{,y} - 0 to y

group - grouping of charachters that can have expressions applied

pattern = r"ex(group)*"

will match ex → 0 or more group

- access matches with .group()

- can be nested

- named groups can be accessed with group() by name
- non-capturing not accessible by group()

r = "(?P<first>abc)(?:def)(ghi)"
named, non capture, regular

| - metacharchter for or

pattern = r"gr(a|e)y"

matches gray or grey

special sequence - backslash followed by charachter

pattern = r"(.+) \1"

↑ matches expression of group 1

will match to 2 words that are the same

\d - matches digits [0-9]

\s - matches whitespoce [\t\n\r\f\v]

\w - matches word charachters [a-zA-Z0-9]

upper case will inverse them

pattern = r"(\D+\d)"

matches one or more non digits followed by a digit

\A - matches beginning of a string

\Z - matches end of a string

\b - matches empty string (boundary between words)

pattern = r"\b(cat)\b"

matches cat with word boundaries on both sides

email extraction - info@sololearn.com

pattern = r"([\w\.-]+)@([\w\.-]+)(\.[\w\.]+)"

matches 1+ word charachters, dot, or slash → first part; → domain (no suffix); → domain suffix

Pythonicness and Packaging

import this - access zen of Python

.make code easy and understandable

PEP - Python style

modules - short, lower case

classes - CapWords

variable/function - lower_underscores

constants - CAP_UNDERSCORES

clash names - trailing underscore (class_)

.lines ≤80 charachters

.avoid from module import *

.one statement per line

.functions can be given any number of arguments

```
def function(named, *args)
```

args is a tuple of other arguments

.functions can have default values

```
def function(x, y, food="spam")
```

if food is not given it is set to spam

.functions can have arbitrary named arguments

```
def function(x, y, **kwargs)
```

kwargs is a dictionary with name as keys and values stored to key

.tuples can be unpacked in mutable objects

```
numbers = (1,2,3)
a,b,c = numbers   a=1, b=2, c=3
```

*variable takes all valuables left after unpacking

```
a, b, *c, d = [1, 2, 3, 4, 5]
```

a=1 , b=2, c=[3,4], d=5

Ternary Operator - conditional expression

```
a=7
b=1 if a>=5 else 42
```

conditional expression

- else statements can follow for/while and will be evaluated as long as there is not a break
- else statements can follow try/except and will be evaluated if no errors occur in the try

- _main_ useful for a file that can be imported or run
  - place script code within

```
if __name__ == "__main__":
    script code
```

Good 3rd party libraries

web framework: Django, CherryPy, Flask

web scraping: Beautiful Soup

graphing: matplotlib

Arrays: NumPy (SciPy)

games: pygame (2D), Panda3D (3D)

Packaging - putting written modules in standard format for use by others

- uses modules: setuptools, distutils

Example Directory

```
SoloLearn/
    License.txt
    Readme.txt
    setup.py          ✱ necessary (import file)
    sololearn/
        __init__.py   ✱ necessary (can be blank)
        sololearn.py  } scripts
        sololearn2.py }
```

Setup File

```
from distutils.core import setup

setup(
    name = 'SoloLearn',
    version = '0.1 dev',
    packages = ['sololearn',],
    license = 'MIT',
    long_description = open('Readme.txt').read(),
)
```

- can now upload to PyPI or create binary distibution
- build source distribution - command line: python setup.py sdist
  python setup.py bdist
  -or-
  (windows) python setup.py bdist_wininst
- upload a package: python setup.py register
  python setup.py sdist upload
- install a package: python setup.py install

- make a non-python executable
  py2exe, Py Installer, cx_Freeze