Python for Beginners: Zero to Knowing Guide

Created by: Josh Wenner

Variables & Basic Data Types

A variable is literally a word that holds a value. The value can be any "Type" of data

```
"Hello, World!" -- String (text)
24 -- Integer
18.5 -- Float
True/False -- Boolean
```

```
my_name = "John"
city = "Berlin"
about_me = my_name + " lives in " + city
print(about_me) -- John lives in Berlin
```

While Loops

while something is True, repeat the code block

Basic Loop with a Counter Variable

```
goal = 0
while goal <= 3:
    print("Score: ", goal )
    goal += 1
```

Allowing a User to Quit the Loop

```
start = input("1 - Start, 2 - Stop")
while start != "2":
    print("Hello and Welcome!")
    start = input("1 - Start, 2 - Stop")
```

User Input

Python can ask users for an input. The value entered can be stored as the "value" to a variable

```
location = input("Enter you region: ")
print("Location: " + location )
```

```
int() -> converts string to integer
float() -> converts string to a decimal

age = int(input("Enter your age: "))
cost = float(input("Enter the cost of the item: "))
print(age + 5)
```

Functions

Reusable blocks of code you create. Data given to a function is called, an **argument**. Data received is called a **Parameter**

Basic function

```
def print_info():
    print("Hello user!")
print_info()
```

Function with an Argument

```
def basic_info(name):
    print("Hello" , name)

basic_info("James")
```

Default Parameter Values

```
def sleep(time="night"):
    print("Bedtime:" , time)

sleep()
sleep("morning")
```

Returning a Value

```
def new_age(age):
    return age - 5

age = new_age(40)
print("New age:", age)
```

***return** allows you to use the value returned as the value to a variable

Function Example

```
def run_test(num):
    if num > 100:
        print("Optimal")
    else:
        print("Used battery")
```

Conditional Statements

Literal Translation -> if something is True, do this. Otherwise if it's not True, do this instead

Conditional Operators

```
equals -- age == 21
not equal -- age != 21
greater than -- age > 21
less than -- age < 21
greater or equal -- age >= 21
less or equal -- age <= 21
```

Working with Lists

```
"thailand" in countries
"spain" not in countries
```

Boolean Values

```
expensive = True
cheap = False
```

Conditional Statement

```
if age < 18:
    print("You are a minor")
elif age <= 21:
    print("You can drink")
else:
    print()
```

Logical Operators

and -- both must be True
or -- only one must be True

```
res = age > 21 and age < 75
sale = age < 18 or age > 70
```

For Loop

for every element in something, I want to do something with that element. Used to Iterate through something

Basic For Loop with Condition

```
message = "Hello"
for letter in message:
    print("-" , letter)  →
```

Output in Terminal

```
-H
-E
-L
-L
-O
```

Looping through a List

```
ages = [24, 32, 55, 65, 45]
for age in ages:
    if age < 18:
        print("Under 18...")
```

→ Checks every number in the list **ages**

Basics of Data Structures

Lists, Tuples, Dictionaries, Sets

Lists

A list is **Mutable** and **Ordered**. You can access elements in a list by indexing the position

Make a List

```
ages  = list() or
ages = [45, 26, 29, 16, 55]
```

Get an element from a list

```
print(ages[1])  →  Output: 26
```

Get the last element in a List

```
print(ages[-1])  →  Output: 31
```

Adding & Removing elements

```
ages = [45, 26, 29, 16, 55]
ages.append(37)   →  [45, 26, 29, 16, 55, 37]
ages.remove(45)   →  [26, 29, 16, 55, 37]
```

Additional list methods

```
ages.sort()         →  [16, 26, 29, 37, 55]
print( len(ages) )     5

extra = (18, 21, 55)
ages.extend(extra)  →  [16, 26, 29, 37, 55, 18, 21, 55]

even = []
for age in ages:
    if age % 2 == 0:
        even.append(age)
print(even)         →  [16, 26, 18]
```

List Comprehensions

Using a Loop to create a list based on a range of numbers

Our loop for even numbers

```
even = []
for age in ages:
    if age % 2 == 0:
        even.append(age)
```

Comprehension for even numbers in list ages

```
age = [ age for age in ages if age % 2 == 0 ]
```

Use a Loop to Capitalize every name

```
names =["jane", "billy", "lily", "tom"]
capital_names = []
for name in names:
    capital_names.append(name)
```

Comprehension for names

```
capital_names = [ name.capitalize() for name in names ]
```

A Single Element in the List

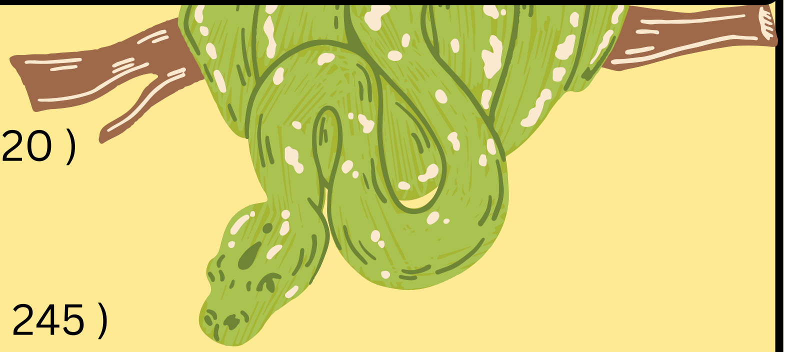
When/Why something is added to the list

Tuples - Immutable

Essentially a List, but you can't change the values

Create a Tuple

```
car_color = tuple()
car_color = ( 240, 15, 20 )
```



Overwrite tuple

```
car_color = ( 50, 100, 245 )
print( car_color[2] )
```

Dictionaries

Every element is stored as a key-value pair - key : value

Make a Dictionary

```
user = dict() or
user = {"a" : 5 , "b" : 10}
```

Getting a value from a dictionary

***dictionary key **unlocks** a Value

***dictionary[key] = value

```
print(user["a"])    →  Output: 5

rank = user.get("b")
print(rank)         →  Output: 10
```

Modify / Delete / Add a Key-Pair

```
user["b"] = 5     →  {"a" : 5 , "b" : 5}
del user["a"]     →  {"b" : 5}
user["c"] = 20    →  {"b" : 5 , "c" : 20}
```

Loop through all Key-Value Pairs

```
for key, value in user.items():     b <-> 5
    print( key, "<->",value)    →   c <-> 20

print( user.keys() )    →  Output: b, c
print( user.values() )  →  Output: 5, 20
```

Nesting Data Structures

```
user = { "tim" :
            { "age" : 26,
              "hobbies" : ["ski","scuba"] }
        }
print( user["tim"]["age"] )  →  Output: 26
user["tim"]["hobbies"].remove("ski")→ ["scuba"]
```

Data Structure - Sets

A collection of Data that is **unordered**, **immutable**, and **unindexed**. No duplicates allowed

Create a Set

```
categories = set()
categories = {"a","e","i", "o","u" }
```

Add / Delete elements

```
categories.add("y")
categories.remove("u")
```

*A set does not allow duplicates

```
ages = [18, 25, 45, 45, 16, 25, 25, 21]
my_set = set(user) ──────→ {18, 25, 45, 16, 21}
```

Combine two sets

```
set1 = { 1, 3, 5, 7, 9, 0 }
set2 = { 2, 3, 4, 5, 6, 7, 8 }
new_set = set1.union(set2) ──→ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

New set based on similarities

```
set1 = { 1, 3, 5, 7, 9, 0 }
set2 = { 2, 3, 4, 5, 6, 7, 8 }
new_set = set1.intersection(set2)────→ {3, 5, 7}
```

New set based on differences

```
set1 = { 1, 3, 5, 7, 9, 0 }
set2 = { 2, 3, 4, 5, 6, 7, 8 }
new_set = set1.difference(set2) ────→ {0, 1, 9}
```

List - Mutable and Ordered
Tuple - Immutable and Unordered
Sets - unordered and unindexed. No duplicate members
Dictionary - Ordered and changeable. Key-Value Pairs

Classes -Object-Oriented Programming

A Class holds Functions (methods) and Variables(properties) which relate to certain types of Objects -- class Animal, Objects - dog, cat, bird

Defining a Class Car

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def car_info(self):
        print("Car Make:", self.make)
        if self.year <= 2000:
            print("The car is old!")
        else:
            print("Car is modern")

porsche = Car("porsche", "911", 2020)
toyota = Car("toyota", "camry", 1998)

porsche.car_info()
toyota.car_info()
```

*****__init__ - init - initialize - start**

*This is a special "Dunder" method that automatically runs when a new object is created.

* __init__ is a place that stores all the properties our class will use throughout

*A method/property must be linked to an object to work

Method - A function in a class
Property - A variable in a class

→

Class inheritance

When one Class, inherits another class. It automatically takes all the properties/methods from the parent class

Defining a a Child Class w/ its own properties

```
class Ferrari( Car ):
    def __init__(self, make, model, year, price):
        super().__init__(make, model, year)
        self.price = price

    def price_check(self, cost):
        if cost >= self.price:
            print("Within Price Range!")
        else:
            print("Outside Price Range!")

italia = Ferrari("Ferrari", "458 italia", 2018, 250000)
italia.car_info()
italia.price_check(200000) ────→ Outside Price Range
```

***super() allows you to inherit/use the superclass

Defining a Class with no new properties

```
class Ferrari( Car ):
    def sell_car(self):
        print("You have sold the", self.year , "Ferrari")

italia = Ferrari("Ferrari", "458 italia", 2018)
italia.car_info()
italia.sell_car()
```

Editing properties

```
italia.model = "Enzo"
italia.year = 1975
```

***A Class can have many different Objects

CLICK HERE TO LEARN PYTHON

Working with Files

Read and write files - text, json, etc.

Reading a file

```
with open("example.txt", "r") as file:
    for line in file:
        print(line)
```

Writing a file (new file or clear)

```
with open("example.txt", "w") as file:
    file.write("This is a new file")
```

Appending/Editing an existing file

```
with open("example.txt" , "a") as file:
    file.write("\nI am adding to a file")
```

Closing the file

```
file.close()
```

Basics of Error Handling

Try to do this, if it fails, do this instead

Nesting Expection Statements

```
try:
    file = open("notes.txt")
    try:                          ←Try to do this
        file.write("Subscribe!")
    except:                       ←If I get an error
        print("Unable to write")
    finally:                      ←No matter the outcome
        file.close()
except FileNotFound:              ←If initial try, fails
    print("File not found...")
else:                             ←If no errors occurred,
                                   run this
    print(“No errors were raised”)
```

New Project

First Repository Setup with GitHub

Prepare GitHub

1. Create a new repo on GitHub or open existing
2. Copy repository HTTPS url found inside
3. Open Terminal in IDE

Git Terminal Steps

1. git init
2. git clone <repo HTTPS url>
3. cd inside the cloned folder
4. git branch <branch-name>
5. git checkout <branch-name>

Congrats, you’re on a new branch within your repository.

Upload Code to GitHub

1. git add - A (or git add .)
2. git commit -m “type note here”
3. git push origin <branch-name>
4. Return to GitHub, Approve the Pull Request

Keeping up-to-date with Changes

1. git fetch origin
2. git status (Recent changes, check your git status)
3. git log origin/main
4. git merge origin/main (Combines current branch with main)

Alternative Option (Only update local repository)

1. git pull origin

***Read and use the commands on the right

Anytime you need help with git you have two commands

git <command> -help ← See options for specific command

git help --all ← Check all possible options

Intro to the Basics of Git

Git Terminal Commands to get started today

git init	→	#Activate git
git clone <url>	→	#Copy Git Repo to local system
git status	→	#Show modified files in current directory
git log	→	#View current commit history
git add -A	→	#Add changed files to your next commit
git commit -m “your message”	→	#commit your changes with a message
git pull origin main	→	#Get up to data changes from main branch
git push origin main	→	#Push your changes to the main branch
git merge my_test	→	#Will merge my_test into main branch
git branch <branch-name>	→	#Will create a new branch
git checkout <branch-name>	→	#switch from the current branch to new branch
git branch -m <new-branch-name>	→	#Will rename current branch
git branch -d <branch-name>	→	#Delete a specfic branch
git rm <file-name>	→	#Remove file from project and stage removal
git stash	→	#Save modified and staged changes
git rebase <branch-name>	→	#Puts commits of current branch ahead of <branch-name>

The Nerd Nook