**Python Basics**

# Variables

## Strings

* Everything inside of quotes
* F-strings is a format that replaces the name of any variable in braces with its value

## Numbers

* Exponentials are represented by two multiplication symbols
* Parentheses to modify the order
* Integers are integers
* Numbers with a decimal point are called Floats
* To make numbers more readable you can use \_ (1\_000\_000)
* You can define multiple variables at once (x,y,z=0,0,0)
* If you want to use variables as constants, make the name if it in capital letters

## Comments

* # indicates a comment

# Lists

* Collection of items in a particular order
* Square brackets indicate a list [], individual items are separated by commas

## Accessing Elements in a list

* Print element by telling the place in the list
* Index numbers start at 0, Not 1
* Last item in the list can be accessed by -1
* You can use f-strings to create a message based on a value gtom a list

bicycles = ['trek', 'cannondale', 'redline', 'specialized']  
➊ message = f"My first bicycle was a {bicycles[0].title()}."

## Changing, Adding, and Removing Elements

* Use the name of the list, followed by the index you want to change and then provide the new value you want the item to have
* Append() method adds elements to the list

motorcycles.append('ducati')  
   print(motorcycles)

* Inserting elements with command insert (index,’…’)

motorcycles.insert(0, 'ducati')  
   print(motorcycles)

* Removing items from the list with command del

del motorcycles[0]

* The pop() method removes the last item in a list, but it lets you work with that item after removing it; use index for choosing specific item
* Removing item by value, use remove method

motorcycles.remove('ducati')

* + Removes only the first occurrence of the value you secify

## Organizing a list

* Sorting with sort() method alphabetically

cars.sort()

* Reverse for sorting reverse

cars.sort(reverse=True)

* Sorting a list temporarily with the sorted() function
* Finding the length of a list with len()

# Working with lists

## Looping through an entire list

* Allows to take same action, or set of actions, with every item in a list
* *For* … *in* …:

*Print…*

* Every lines of code after the for loop that are not intented are executed once without repetition

## Using range() Function

* Makes it easy to create a series of numbers

For value in range (1,5):

Print (value)

* Range function includes the off-by-one behavior (prints numbers 1-4)
* You can pass range() one argument: range(6) would return the numbers from 0-5
* To make a list of numbers, convert result of range into a list

Numbers=list(range(1,6))

Print(numbers)

* Range() function can be used to skip numbers by adding third argument: This argument is giving the step size when generating numbers

## Simple statistics with a List of Numbers

* Minimum: min()
* Maximum: max()
* Sum: sum()

## List comprehension

* Allows to create the same list in just one line of code
* Combines the for loop and the creation of new elements into one line, and automatically appends each new element

squares=[value\*\*2 for value in range(1,11)]

print(squares)

# Working with part of a list

## Slicing a list

* Specify the first and last item in the list you want to work with

Print(…[Anfangsindex:Endindex])

* Third item indicates which one to skip

## Looping through a slice

* Instead of looping through the entire list, python is only looping through the slice

print("Here are the first three players on my team:")  
➊ for player in players[:3]:  
       print(player.title())

## Copying a list

* To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index ([:])
* Enter key: \n
* It is important to tell python that it is a slice, otherwise it will always be the same

## Tuples

* List of items that cannot change “immutable”
* Tuples look just like a list except you use parentheses instead of square brackets
* dimensions = (200, 50)  
   print(dimensions[0])  
     print(dimensions[1])
* If you want a tuple with one element, you need a trailing comma
* Redefine entire tuple for changes

# IF Statements

# Functions

* Blocks of code designed to do one specific job
* Define function

def greet\_user():  
➋     """Display a simple greeting."""  
➌     print("Hello!")  
  
➍ greet\_user()

## Arguments and Parameters

* Parameter is a piece of function that the function needs to do the job
* An argument is a piece of information that’s passed from a function call to a function

def greet\_user(username):  
    """Display a simple greeting."""  
    print(f"Hello, {username.title()}!")  
  
greet\_user('jesse')

* Username is parameter and ‘jesse’ is argument

# Passing Arguments

## Positional arguments

* Python must match each argument in the function call with a parameter in the function definition

def describe\_pet(animal\_type, pet\_name):  
       """Display information about a pet."""  
       print(f"\nI have a {animal\_type}.")  
       print(f"My {animal\_type}'s name is {pet\_name.title()}.")

* The order matters

## Keyword arguments

* Name-value pair that you pass to a function
* You directly associate the name and the value within the argument
* The order does not matter anymore

## Default values

* If no argument is called, python uses the default value
* Default values have to be defined before non-default values

## Equivalent function calls

* There are multiple ways to code a function

## Return values

def get\_formatted\_name(first\_name, last\_name):  
       """Return a full name, neatly formatted."""  
➋     full\_name = f"{first\_name} {last\_name}"  
➌     return full\_name.title()

➍ musician = get\_formatted\_name('jimi', 'hendrix')  
   print(musician)

## Making an argument optional

* Using the function can choose to provide extra information
* Set default value to an empty string and move it to the end of the list of parameters
* Middle name has to be last argument then

## Using a Function with a while Loop

## Passing a list

* def greet\_users(names):  
         """Print a simple greeting to each user in the list."""  
         for name in names:  
             msg = f"Hello, {name.title()}!"  
             print(msg)
* ➊ usernames = ['hannah', 'ty', 'margot']  
     greet\_users(usernames)

## Modifying a list in a function

* Function can modify a list

# Classes and Objects

* Objects are an encapsulation of variables and functions
* Classes essentially a template to create your objects
* You can assign a class to an object
* Accessing object functions is similar

class MyClass:

variable = "blah"

def function(self):

print("This is a message inside the class.")

myobjectx = MyClass()

myobjectx.function()

# Dictionaries

* Data type similar to arrays, but works with keys and values instead of indexes

phonebook = {}

phonebook["John"] = 938477566

phonebook["Jack"] = 938377264

phonebook["Jill"] = 947662781

print(phonebook)

print(Jack)

* Alternative

phonebook = {

"John" : 938477566,

"Jack" : 938377264,

"Jill" : 947662781

}

print(phonebook)

* Iterating over dictionaries

phonebook = {"John" : 938477566,"Jack" : 938377264,"Jill" : 947662781}

for name, number in phonebook.items():

print("Phone number of %s is %d" % (name, number))

* Removing a value in a dictionary:
  + Use del …[]
  + Or use….pop[]

## Modules and packages

* Module is a piece of software that has a specific functionality
* Modules in python are files with a .py extension
* Module can have a set of functions, classes or variables
* Importing module objects

from draw import draw\_game

or

import draw.draw\_game

def main():

result = play\_game()

draw\_game(result)

* Importing all objects

from draw import \*

def main():

result = play\_game()

draw\_game(result)

| **Argument** | **Description** |
| --- | --- |
| path | String indicating filesystem location, URL, or file-like object |
| sep or delimiter | Character sequence or regular expression to use to split fields in each row |
| header | Row number to use as column names; defaults to 0 (first row), but should be None if there is no header row |
| index\_col | Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index |
| names | List of column names for result, combine with header=None |
| skiprows | Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip. |
| na\_values | Sequence of values to replace with NA. |
| comment | Character(s) to split comments off the end of lines. |
| parse\_dates | Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns). |
| keep\_date\_col | If joining columns to parse date, keep the joined columns; False by default. |
| converters | Dict containing column number of name mapping to functions (e.g., {'foo': f} would apply the function f to all values in the 'foo' column). |
| dayfirst | When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); False by default. |
| date\_parser | Function to use to parse dates. |
| nrows | Number of rows to read from beginning of file. |
| iterator | Return a TextParser object for reading file piecemeal. |
| chunksize | For iteration, size of file chunks. |
| skip\_footer | Number of lines to ignore at end of file. |
| verbose | Print various parser output information, like the number of missing values placed in non-numeric columns. |
| encoding | Text encoding for Unicode (e.g., 'utf-8' for UTF-8 encoded text). |
| squeeze | If the parsed data only contains one column, return a Series. |
| thousands | Separator for thousands (e.g., ',' or '.'). |