Day - 4

**Randomization**

**& Lists**

Random number generators and lists.

**4.1 import, modules**

Modules are similar as other languages, for code management. For larger code we use separate files. These separate **.py** files are called modules. To import a module we use

**import** module\_name

* For example, to import random module we use: **import** random

**4.2 Integer and float random number generators**

**import** random

#Integer random:*following generate random int between 1 and 10,* including 1 & 10

random\_int = random**.randint**(1, 10)

**print**(f"int random number between 1 to 10 is : {random\_int}")

#Float random:*following generate random float between 0 and 1 i.e (0, 1)*

random\_float = random**.random**()

**print**(f"float random number between 0 to 1 is : {random\_float}")

#*follwoing gives float*randoms betwwen 0 and 4

**print**(f"random number between 0 to 5 is : {random**.random**()\*5}")

#*Modified*Lovecalculator*: Generates random number*from 1 to 100, including 1 & 100

love\_score = random**.randint**(1, 100)

**print**(f"Your love score is {love\_score}")

#*python rand\_intro.py*

* Exercise 4.1: Coin toss game

**import** random

coin\_rand = random**.randint**(0, 1)

**if** coin\_rand **==** 1 :

**print**("\n\n\t\tHEAD")

**else** :

**print**("\n\n\t\tTAIL")

#*python coin.py*

|  |  |
| --- | --- |
| **4.3 list**  A built-in Python ***sequence***. Despite its name it is more akin to an ***array*** in other languages than to a ***linked*** ***list*** since access to elements is O(1).   * Don’t need to memorize all of these method, just Google what you need. |  |

#*List is a****Data******Structure****in Python*

#*It stores****list****of data as : [otem1, item2, . . . ].*

#*It is used for****Grouping****and****Ordering****data*

fruits = ["Cherry", "Apple", "Pear"]

#*the items vcan be*accessed/modified*just like array.*

**print**(fruits[2])

fruits[2] = "Havana"

**print**(fruits[2])

#*You can also use*negative index*. To access items from the END*

#*"-1" indicates last item. So no "-0"*

**print**(fruits[-1])

#Adding *new items. Index doesn't matter*

fruits**.append**("Jackfruit")

**print**(fruits[-1])

#Extending *list*

fruits**.extend**(["Mango", "Banana", "Pineapple"])

**print**(fruits)

* Exercise 4.2.1: Banker Roulette game.

**import** random

names = ["Angela", "Ben", "Jenny", "Michael", "Chole"]

length = **len**(names)

ran = random**.randint**(1, length)

**print**(f"Bill will be fulfilled by {names[ran-1]}")

#*python bnkrulet.py*

**4.4 split**

s.**split**(t, n)

Returns a ***list*** of ***strings*** splitting at most ***n*** times on str ***t***; if ***n*** isn’t given, splits ***as many times as possible***; if t isn’t given, splits on ***whitespace***. Use ***str.rsplit()*** to split from the right—this makes a difference only if n is given and is less than the maximum number of splits possible. For example, following will split he string by appearance of ", " comma and white space

data = **input**("Enter the names : ")

names = data**.split**(", ")

* Exercise 4.2.2: Banker Roulette game with user input:

**import** random

data = **input**("Enter the names : ")

#splitting*string*

names = data**.split**(", ")

**print**(f"Your data is: {names}")

length = **len**(names)

ran = random**.randint**(1, length)

**print**(f"Bill will be fulfilled by {names[ran-1]}")

#*python bnkrulet.py*

* Exercise 4.2.2: Banker Roulette game with user input. Using **random.choice()** method of **random** module:

**import** random

data = **input**("Enter the names : ")

#*splitting string*

names = data**.split**(", ")

**print**(f"Your data is: {names}")

#*using random.choice*

ran\_name = random**.choice**(names)

**print**(ran\_name + " Will pay the bill today")

#*python bnkrulet.py*

**4.5 IndexError and Nested lists**

Index error is thrown if the index does not exist. For fruit = ["Cherry", "Apple", "Pear"] then ***fruit[3]*** will throw an index error. Because ***fruit[3]*** doesn't exist.

* Nested list: List could be nested inside one another. Following demonstrates this:

#*Nested lists*

fruit = ["Cherry", "Apple", "Pear"]

veg = ["Potato", "Tomato", "Carrot"]

#*following is*nested*list*

buy\_list = [fruit, veg]

**print**(buy\_list)

#*Output: [['Cherry', 'Apple', 'Pear'], ['Potato', 'Tomato', 'Carrot']]*

* Exercise 4.3: Treasure map game. Change selected element of the nested array.

row1 = ["o", "o", "o"]

row2 = ["o", "o", "o"]

row3 = ["o", "o", "o"]

map = [row1, row2, row3]

**print**(f"{row1}\n{row2}\n{row3}")

position = **input**("Where you want to put your treasure ? : row-col ")

slct\_row = map[**int**(position[0])-1]

slct\_row[**int**(position[1])-1] = "x"

#*or we can*accessdirectly*as 2-d array as a[m][n]*

**print**(f"{map[1][2]}")

hrz = **int**(position[0])-1

vert = **int**(position[1])-1

map[hrz][vert] += "D"

**print**(f"{row1}\n{row2}\n{row3}")

#*python trasure\_map.py*

* Exercise 4.4: Rock Paper Scissor game map game.

**import** random

#*Rock Paper Scissors ASCII Art*

rock = """

    \_\_\_\_\_\_\_

---'   \_\_\_\_)

      (\_\_\_\_\_)

      (\_\_\_\_\_)

      (\_\_\_\_)

---.\_\_(\_\_\_)

"""

scissors = """

    \_\_\_\_\_\_\_

---'   \_\_\_\_)\_\_\_\_

          \_\_\_\_\_\_)

       \_\_\_\_\_\_\_\_\_\_)

      (\_\_\_\_)

---.\_\_(\_\_\_)

"""

paper = """

     \_\_\_\_\_\_\_

---'    \_\_\_\_)\_\_\_\_

           \_\_\_\_\_\_)

          \_\_\_\_\_\_\_)

         \_\_\_\_\_\_\_)

---.\_\_\_\_\_\_\_\_\_\_)

"""

user= **int**(input("Choose 1 for Rock, 2 for Scissor, 3 for Paper : "))

pc\_rand = random**.randint**(1, 3)

element = [rock, scissors, paper]

**if** (user **>** 2) **or** (user **<** 0):

**print**("Invalid Number. PC wins")

**elif** pc\_rand **==** user :

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tDRAW")

**elif** (pc\_rand **==** 1) **and** (user **==** 2):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tPC wins")

**elif** (pc\_rand **==** 1) **and** (user **==** 3):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tUser wins")

**elif** (pc\_rand **==** 2) **and** (user **==** 1):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tUser wins")

**elif** (pc\_rand **==** 2) **and** (user **==** 3):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tPC wins")

**elif** (pc\_rand **==** 3) **and** (user **==** 2):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tUser wins")

**elif** (pc\_rand **==** 3) **and** (user **==** 1):

**print**(f"\n\tYou Chosed :\n{element[user-1]}\n\n\tPC chosed :\n {element[pc\_rand-1]}\n\n\t\tPC wins")

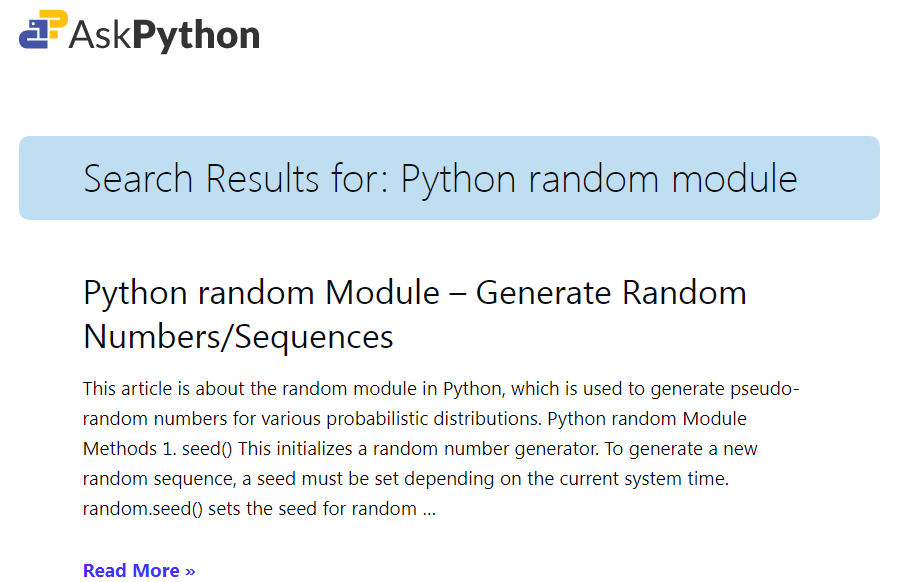
#*python rc\_pap\_sci.py*

**Appendix:** Python random Module – Generate Random Numbers/Sequences

Mersenne Twister: The Mersenne Twister is a pseudorandom number generator (PRNG). It is by far the most widely used general-purpose PRNG. Its name derives from the fact that its period length is chosen to be a ***Mersenne*** ***prime***.

More on Khan Academy.

<https://www.askpython.com>



**4.6** appendix **Python random Module Methods**

1. seed()

This initializes a random number generator. To generate a new random sequence, a seed must be set depending on the current system time. ***random.seed()*** sets the seed for random number generation.

1. getstate()

This returns an object containing the current state of the generator. To restore the state, pass the object to ***setstate()***.

1. setstate(state\_obj)

This restores the state of the generator at the point when ***getstate()*** was called, by passing the state object.

1. getrandbits(k)

This returns a Python integer with k random bits. This is useful for methods like***randrange()*** to handle arbitrary large ranges for random number generation.

**4.7** appendix **Generate Random Integers**

The random module provides some special methods for generating random integers.

### randrange(start, stop, step)

Returns a randomly selected integer from ***range(start, stop, step)***. This raises a ***ValueError*** if ***start > stop***.

### randint(a, b)

Returns a random integer between **a** and **b** (both inclusive). This also raises a ***ValueError***if ***a > b***.

Here is an example that illustrates both the above functions.

**4.8** appendix **Generating Random floating point numbers**

Similar to generating integers, there are functions that generate random floating point sequences.

* random.**random**() -> Returns the next random floating point number between [0.0 to 1.0)
* random.**uniform**(a, b) -> Returns a random floating point N such that if and  if b < a.
* random.**expovariate**(lambda) -> Returns a number corresponding to an exponential distribution.
* random.**gauss**(mu, sigma) -> Returns a number corresponding to a gaussian distribution.
* There are similar functions for other distributions, such as Normal Distribution, Gamma Distribution, etc.

**4.9** appendix **Random Sequences using the random module**

Similar to integers and floating-point sequences, a generic sequence can be a collection of items, like a List / Tuple. The ***random*** module provides useful functions which can introduce a state of randomness to sequences.

1. **random.shuffle(x)**

This is used to shuffle the sequence in place. A sequence can be any list/tuple containing elements. Example Code to illustrate shuffling:

**import** random

sequence = [random**.randint**(0, i) **for** i **in** **range**(10)]

#*Notice the '+'  not used. Instead the output is sent as parameter*

**print**('Before shuffling the random sequence : ', sequence)

#*Applying shuffle()*

random**.shuffle**(sequence)

**print**('After shuffling', sequence)

#*python ranmeth.py*

|  |
| --- |
|  |

Possible Output:

Before shuffling [0, 0, 2, 0, 4, 5, 5, 0, 1, 9]

After shuffling [5, 0, 9, 1, 5, 0, 4, 2, 0, 0]

1. **random.choice(seq)**

This is a widely used function in practice, wherein you would want to randomly pick up an item from a List/sequence.

**import** random

a = ['one', 'eleven', 'twelve', 'five', 'six', 'ten']

**print**(a)

**for** i **in** **range**(5):

**print**(random**.choice**(a))

|  |
| --- |
|  |

Possible Output

['one', 'eleven', 'twelve', 'five', 'six', 'ten']

ten

eleven

six

twelve

twelve

|  |
| --- |
|  |

1. **random.sample(population, k)**

Returns a random sample from a sequence of length **k**.

**import** random

a = ['one', 'eleven', 'twelve', 'five', 'six', 'ten']

**print**(a)

**for** i **in** **range**(3):

    b = random**.sample**(a, 2)

**print**('random sample:', b)

|  |
| --- |
|  |

Possible Output

['one', 'eleven', 'twelve', 'five', 'six', 'ten']

random sample: ['five', 'twelve']

random sample: ['ten', 'six']

random sample: ['eleven', 'one']

|  |
| --- |
|  |

**4.10** appendix **Random Seed**

Since pseudorandom generation is based on the previous number, we usually use the system time to make sure that the program gives a new output every time we run it. We thus make use of ***seeds***.

Python provides us with ***random.seed()*** with which we can set a seed to get an initial value. This seed value determines the output of a random number generator, so if it remains the same, the output also remains the same.

|  |
| --- |
|  |

**import** random

random**.seed**(1)

**print**('Generating a random sequence of 4 numbers...')

**print**([random**.randint**(1, 100) **for** i **in** range(5)])

#*Reset the seed to 1 again*

random**.seed**(1)

#*We now get the same sequence*

**print**([random**.randint**(1, 100) **for** i **in** range(5)])

Possible Output

Generating a random sequence of 4 numbers...

[18, 73, 98, 9, 33]

[18, 73, 98, 9, 33]

|  |
| --- |
|  |

This ensures that we need to be mindful of our seed when dealing with pseudorandom sequences, since the sequence may repeat if the seed is unchanged.