Built in data types

Antra

Native Datatypes in Python

- Python is **Dynamically** and **Strongly typed**
- You don’t need to explicitly declare the data type of variables in python
- Python will “guess” the datatype and keeps track of it internally.

Native Datatypes in Python

- **Booleans** are either True or False.
  - `x = True`
- **Numbers** can be integers (1 and 2), floats (1.1 and 1.2), fractions (1/2 and 2/3), or even complex numbers.

```
x = 20
```

```
x = 20.5
```

- **Strings** are sequences of Unicode characters, e.g. an html document.

```
x = "Hello World"
```

Native Datatypes in Python

- **Bytes** and **byte arrays**, e.g. a jpeg image file.

```
x = b"Hello"
```

```
x = bytearray(5)
```

- **Lists** are ordered sequences of objects.

```
x = ["apple", "banana", "cherry"]
```

- **Tuples** are ordered, **immutable** sequences of values.

```
x = ("apple", "banana", "cherry")
```

Native Datatypes in Python

- **Sets** and **Frozensets** are unordered bags of values.

```
x = {"apple", "banana", "cherry"}
```

```
x = frozenset({"apple", "banana", "cherry"})
```

- **Dictionaries** are ordered bags of key-value pairs.
  - In versions before 3.7, dicts are either not ordered or the order is not guaranteed
  - Starting in 3.7, elements in dicts are guaranteed to be in the order they are inserted

```
x = {"name" : "John", "age" : 36}
```

Operators

Antra

Basic Operators in Python

- Arithmetic
  - Exponentiation: **
  - Floor division: //
- Assignment
  - x = x +1 => x += 1 (Note -- and ++ are not supported in Python)
- Comparison
  - Not equal: !=
- Logical operators
  - And
  - or
  - Not

Basic Operators in Python

- Identity
  - Check if two objects are the same object, with the same memory location
    - Is
    - is not
- Membership
  - Check if a sequence is presented in an object
    - in
    - not in

Functions

Antra

Multiple Return Values

- Python functions can return multiple values using one return statement.

```
def square_point(x, y, z):
  x_squared = x * x
  y_squared = y * y
  z_squared = z * z
  # Return all three values:
  return x_squared, y_squared, z_squared

three_squared, four_squared, five_squared
= square_point(3, 4, 5)
```

Discarding Unwanted values

- When returning multiple values, you are really returning a tuple, so you can use indexing, slicing to only get the values you want.
-

```
def discard_middle_value():
  return 1,2,3
def only_use_last_value():
  return 1,2,3

# this is okay, but make sure nothing else is called "_"
a,_,c=discard_middle_value()
print(a,c)
print(a,_,c)

d,e = discard_middle_value()[0:3:2]
print(d,e)
f = only_use_last_value()[2]
print(f)
```

```
1 3
1 2 3
1 3
3
```

Variable Scope

- A variable defined inside a function has local scope, a variable defined outside a function has global scope.
- Variables with local scope can only be used within the function its defined in
- If two variables with the same name are defined both inside and outside of an function, they won't interfere with each other.
- Note if a variable is only defined in the global scope, functions can still use it.

```
a = 5

def f1():
  a = 2
  print(a)

print(a)   # Will print 5
f1()       # Will print 2
```

```
a = "Hello"

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

Loops with range()

- In Python, a for loop can be used to perform an action a specific number of times in a row.
- The range() function can be used to create a list that can be used to specify the number of iterations in a for loop.

```
# Print the numbers 0, 1, 2:
for i in range(3):
  print(i)

# Print "WARNING" 3 times:
for i in range(3):
  print("WARNING")
```

Looping Through Dictionaries

- You can use a for loop to loop through a dictionary in Python
- You can either loop through just the keys, or both the keys and values, you can even use .item() to loop a dictionary as tuples

```
>>> for key, value in a_dict.items():
...     print(key, '->', value)
...
color -> blue
fruit -> apple
pet -> dog
>>> for key in a_dict:
...     print(key, '->', a_dict[key])
...
color -> blue
fruit -> apple
pet -> dog
```

```
>>> for item in a_dict.items():
...     print(item)
...     print(type(item))
...
('color', 'blue')
<class 'tuple'>
('fruit', 'apple')
<class 'tuple'>
('pet', 'dog')
<class 'tuple'>
```

while Loops

- In Python, a while loop will repeatedly execute a code block as long as a condition evaluates to True.
- The condition of a while loop is always checked first before the block of code runs. If the condition is not met initially, then the code block will never run.

```
# This loop will only run 1 time
hungry = True
while hungry:
  print("Time to eat!")
  hungry = False

# This loop will run 5 times
i = 1
while i < 6:
  print(i)
  i = i + 1
```

break Keyword

- break keyword escapes the loop, regardless of the iteration number.
- Note, in nested loops, break will only break out of the inner loop.

```
numbers = [0, 254, 2, -1, 3]

for num in numbers:
  if (num < 0):
    print("Negative number
detected!")
    break
  print(num)

# 0
# 254
# 2
# Negative number detected!
```

```
for i in range(10):
    print(i)
    for e in range(10, 15):
        print(e)
        if e==11:
            break
```

```
0
10
11
1
10
11
2
10
11
3
```

continue Keyword

- The continue keyword is used inside a loop to skip the remaining code inside the loop code block and begin the next loop iteration.

```
big_number_list = [1, 2, -1, 4,
-5, 5, 2, -9]

# Print only positive numbers:
for i in big_number_list:
  if i < 0:
    continue
  print(i)
```

Control Flow

Antra

Control Flow Keywords in Python

- or
  - Combines two Boolean expressions and evaluates to True if at least one of the expressions returns True. Otherwise, if both expressions are False, then the entire expression evaluates to False.

```
True or True      # Evaluates to True
True or False     # Evaluates to True
False or False    # Evaluates to False
1 < 2 or 3 < 1    # Evaluates to True
3 < 1 or 1 > 6    # Evaluates to False
1 == 1 or 1 < 2   # Evaluates to True
```

Control Flow Keywords in Python

- Elif
  - Short for else if, used in conjunction with if and else

```
pet_type = "fish"

if pet_type == "dog":
  print("You have a dog.")
elif pet_type == "cat":
  print("You have a cat.")
elif pet_type == "fish":
  # this is performed
  print("You have a fish")
else:
  print("Not sure!")
```

Control Flow Keywords in Python

- and
  - Performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to True then the and operator returns True. If either side (or both sides) evaluates to False, then the and operator returns False.
  - A non-Boolean value (or variable that stores a value) will always evaluate to True when used with the and operator.

```
True and True      # Evaluates to True
True and False     # Evaluates to False
False and False    # Evaluates to False
1 == 1 and 1 < 2   # Evaluates to True
1 < 2 and 3 < 1    # Evaluates to False
"Yes" and 100      # Evaluates to True
```

Control Flow Keywords in Python

- not
  - Used in a Boolean expression in order to evaluate the expression to its inverse value.
  - Note Python doesn't support prepending a ! at the front of an Boolean expression. (you can still do !=)

```
not True     # Evaluates to False
not False    # Evaluates to True
1 > 2        # Evaluates to False
not 1 > 2    # Evaluates to True
1 == 1       # Evaluates to True
not 1 == 1   # Evaluates to False
```

```
print(1!=1)   # this is valid
print(!false) # this is not
```

Dictionary

Antra

- Values in a Python dictionary can be accessed by placing the key within square brackets next to the dictionary.
- Values can be written by placing key within [] next to the dictionary and using the assignment operator (=).
- If the key already exists, the old value will be overwritten.
- Attempting to access a value with a key that does not exist will cause a KeyError.

Types in Dictionaries

- The values in a dictionary can be any type
- The keys however, must be an immutable type, such as string, numbers and tuples.

```
d = {"a":1, "b":"apple", "c":[1,2,3], 4:"orange"}

for k,v in d.items():
  print(v,type(v))
```

```
1 <class 'int'>
apple <class 'str'>
[1, 2, 3] <class 'list'>
orange <class 'str'>
```

- You can even store functions in a dictionary

```
def f1():
    print("apple")

def f2():
    print("pear")

def f3():
    print("orange")

d1 = {"a": f1, "p": f2, "o": f3}

for f in d1:
    d1[f]()
```

```
apple
pear
orange
```

```
def add(n,m):
  print(n+m)

def sub(n,m):
  print(n-m)
for k,v in d2.items():
  v(3,4)
```

```
7
-1
```

Accessing Keys and Values in Dictionaries

```
d2 = {"1":add, "2":sub}

print(d2.keys())
print(d2.values())
print(d2.items())
```

```
dict_keys(['1', '2'])
dict_values([<function add at 0x000001EF43054678>, <function sub at 0x000001EF43054558>])
dict_items([('1', <function add at 0x000001EF43054678>), ('2', <function sub at 0x000001EF43054558>)])
```

Retrieve Stored Values in Dictionaries

- You can use **dic[“key_name”]** to retrieve stored values in dictionaries, but when the dictionary doesn’t have such key, you will get an exception.
- Alternatively, you can use dic**.get(“key_name”, default_value)**
  - If the key exists, the corresponding value will be return
  - Else, the default value will be returned
  - If no default value is specified, None will be returned

Removes Items from Dictionaries

- You can use **.pop(key_name)** to remove key-value pairs from dictionaries.
- The corresponding pair will be removed, and the value will be returned.

Merging Dictionaries

- .update() can be used to combine two dictionaries
- The key value pairs in the dictionary used as the parameter will be added to the calling dictionary
- If there are duplicate keys in the calling and the input dictionary, the value in the calling dictionary will be updated

```
d1 = {"a":1, "b":2, "c":3}
d2 = {"c":4, "d":5, "e":6}

d1.update(d2)
print(d1)
```

```
{'a': 1, 'b': 2, 'c': 4, 'd': 5, 'e': 6}
```

Tuple

Antra

- Tuples are similar to Lists but are immutable
- Use var_name = () to create an empty tuple
- Use var_name = “something”, to create an one-tuple
  - Note the “,” at the end
  - Var_name = (“something”) will not work
- You can simply assign multiple values to the same variable to create a tuple

```
a = 1, "E", 3
print(a,type(a))
```

```
(1, 'E', 3) <class 'tuple'>
```

Set

Antra

- Sets are **unordered** collections with **no duplicate** elements.
- Basic uses include membership testing and eliminating duplicate entries.
- You can also perform mathematical operations like union, intersection, difference and symmetric difference on sets.

Set Operations

```
s1 = {'a','b','c','c'} # only one 'c' will show up in the result
s2 =  set('cde')

print(s1, s2)
print(s1 - s2) # items in s1 but not s2
print(s1 | s2) # items in s1 or s2
print(s1 & s2) # items in s1 and s2
print(s1 ^ s2) # items in s1 or s2 but not both
```

```
{'c', 'b', 'a'} {'d', 'c', 'e'}
{'b', 'a'}
{'e', 'b', 'd', 'c', 'a'}
{'c'}
{'b', 'd', 'a', 'e'}
```

String Operations

Antra

Sequence Operations

s2 in s	Return true if s contains s2
s + s2	Concat s and s2
len(s)	Length of s
min(s)	Smallest character of s
max(s)	Largest character of s
s2 not in s	Return true if s does not contain s2
s * integer	Return integer copies of s concatenated # 'hello' => 'hellohellohello'
s[index]	Character at index of s
s[i:j:k]	Slice of s from i to j with step k
s.count(s2)	Count of s2 in s
s[::-1]	Return s reversed

Whitespace

s.center(width)	Center s with blank padding of width # 'hi' => ' hi '
s.isspace()	Return true if s only contains whitespace characters
s.ljust(width)	Left justifiy s with total size of width # 'hello' => 'hello '
s.rjust(width)	Right justify s with total size of width # 'hello' => ' hello'
s.strip()	Remove leading and trailing whitespace from s # ' hello ' => 'hello'
s.center(width, pad)	Center s with padding pad of width # 'hi' => 'padpadhipadpad'
s.expandtabs(integer)	Replace all tabs with spaces of tabsize integer # 'hello\tworld' => 'hello world'
s.lstrip()	Remove leading whitespace from s # ' hello ' => 'hello '
s.rstrip()	Remove trailing whitespace from s # ' hello ' => ' hello'
s.zfill(width)	Left fill s with ASCII '0' digits with total length width # '42' => '00042'

Find/Replace

s.index(s2, i, j)	Index of first occurrence of s2 in s after index i and before index j
s.find(s2)	Find and return lowest index of s2 in s
s.index(s2)	Return lowest index of s2 in s (but raise ValueError if not found)
s.replace(s2, s3)	Replace s2 with s3 in s
s.replace(s2, s3, count)	Replace s2 with s3 in s at most count times
s.rfind(s2)	Return highest index of s2 in s
s.rindex(s2)	Return highest index of s2 in s (raise ValueError if not found)

Case

s.capitalize()	Capitalize s # 'hello' => 'Hello'
s.lower()	Lowercase s # 'HELLO' => 'hello'
s.swapcase()	Swap cases of all characters in s # 'Hello' => "hELLO"
s.title()	Titlecase s # 'hello world' => 'Hello World'
s.upper()	Uppercase s # 'hello' => 'HELLO'
s.casefold()	Casefold s (aggressive lowercasing for caseless matching) # 'ßorat' => 'ssorat'
s.islower()	Return true if s is lowercase
s.istitle()	Return true if s is titlecased # 'Hello World' => true
s.isupper()	Return true if s is uppercase

Inspection

s.endswith(s2)	Return true if s ends with s2
s.isalnum()	Return true if s is alphanumeric
s.isalpha()	Return true if s is alphabetic
s.isdecimal()	Return true if s is decimal
s.isnumeric()	Return true if s is numeric
s.startswith(s2)	Return true is s starts with s2
s.endswith((s1, s2, s3))	Return true if s ends with any of string tuple s1, s2, and s3
s.isdigit()	Return true if all the characters are digits
s.isidentifier()	Return true if s is a valid identifier/valid variable name
s.isprintable()	Return true is s is printable

List and List Operations

Python has no built-in support for Arrays, but Lists are versatile and powerful..

Adding Lists Together

- lists can be added to each other using the plus symbol.
- The result is a new list containing items from both lists.

```
items = ['cake', 'cookie', 'bread']
total_items = items + ['biscuit', 'tart']
print(total_items)
# Result: ['cake', 'cookie', 'bread', 'biscuit',
'tart']
```

Multiple Datatypes in Lists

- Python lists can contain items of different datatypes.
- You can put numbers, string, other objects, even other lists inside a list.

```
numbers = [1, 2, 3, 4, 10]
names = ['Jenny', 'Sam', 'Alexis']
mixed = ['Jenny', 1, 2]
list_of_lists = [['a', 1], ['b', 2]]
```

Adding Items to a List

- You can add values to the end of a list using the **.append()** method.
- The result will be the same list with the new item appended to the end.
- To add values to other locations, you can use **.insert(index, value)**.
  - However, if you want to add to the beginning of the list, consider using deque(double ended queue instead of a list) as adding to the beginning of a list is very inefficient.

```
orders = ['daisies', 'periwinkle']
orders.append('tulips')
print(orders)
# Result: ['daisies', 'periwinkle', 'tulips']
```

Aggregating Iterables with zip()

- Data types that can be iterated (called iterables) can be used with the zip() function to aggregate data.
- The zip() function takes iterables, aggregates corresponding elements based on the iterables passed in, and returns an iterator. Each element of the returned iterator is a tuple of values.

List Slicing

- Python lists are 0 indexed, when slicing from the beginning of the list, the 0 can be omitted.
- Similarly, it's unnecessary to specify the last index when slicing to the end.
- The slice will include the starting index but excluding the ending index.

```
items = [1, 2, 3, 4, 5, 6]

# All items from index `0` to `3`
print(items[:4])

# All items from index `2` to the last item, inclusive
print(items[2:])
```

List Methods

append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list
len()	Returns the length of the list

Negative List Indices

- Negative indices for lists in Python can be used to reference elements in relation to the end of a list.
- This can be used to access single list elements or as part of defining a list range.

```
soups = ['minestrone', 'lentil', 'pho', 'laksa']
soups[-1]   # 'laksa'
soups[-3:]  # 'lentil', 'pho', 'laksa'
soups[:-2]  # 'minestrone', 'lentil'
```

Generating Lists with range()

- range(stop) -> range object range(start, stop[, step]) -> range object
- Return an object that produce(can be turned into an iterator) a sequence of integers from start (inclusive) to stop (exclusive) by step.
  - range(i, j) produces i, i+1, i+2, ..., j-1. start defaults to 0, and stop is omitted!
  - range(4) produces 0, 1, 2, 3. These are exactly the valid indices for a list of 4 elements.
  - When step is given, it specifies the increment (or decrement).

```
my_range = range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
r = range(100, 0, -10)
print(list(r))
```

```
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

```
r = range(100, 0, -10)
print(iter(r))
print(next(iter(r)))
```

```
100
```

```
>>> my_range3 = range(1, 100, 10)
>>> print(list(my_range3))
[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
```

Generators in Python

Antra

- Generator is a very useful Python feature that’s rarely found elsewhere.
- Generators are used to create iterators, thus we need to understand what an iterator is before learning about generators

Iterator

- An object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation.
- An iterator is an object which implements the iterator protocol
- The iterator protocol consists of two methods:
  - __iter__() method, which returns the iterator object.
  - __next__() method, which returns the next element of sequence.

Why make an iterator?

- Iterators allow you to make an iterable that computes its items as it goes.
- Iterators are lazy, in that they don't determine what their next item is until you ask them for it.
- Using an iterator instead of a list, set, or another iterable data structure can sometimes allow us to save memory.
-

```
>>> from itertools import repeat
>>> lots_of_fours = repeat(4, times=100_000_000)
>>> import sys
>>> sys.getsizeof(lots_of_fours)
56
```

```
>>> lots_of_fours = [4] * 100_000_000
>>> import sys
>>> sys.getsizeof(lots_of_fours)
800000064
```

The iterator and list both provides 100 million 4's to us. The iterator takes 56 bytes while the list takes many megabytes of memory.

Iterable Object Example

- We can use for to loop through a list:

```
for i in[1,2,3,4]:
    print(i)
```

```
1
2
3
4
```

- We can loop through the chars in a string:

```
for i in "PythonSEP":
    print(i)
```

```
P
y
t
h
o
n
S
E
P
```

- We can loop through the keys in a dictionary:

```
for i in {"X":1,"Y":2}:
    print(i)
```

```
X
Y
```

- We can even loop through lines in a file like this:

```
for line in open("PythonSEP.txt"):
    print(line)
```

Creating & Consuming an Iterator

- Iterable is anything you're able to loop over; iterator is the object that does the actual iterating.
- The built-in function iter takes an iterable object and returns an iterator.
- Each time we call the .__next__() method on the iterator, we get the next element.
- If there are no more elements, it raises a StopIteration

```
x= iter([1,2,3])
print(x)
print(x.__next__())
print(next(x))
print(x.__next__())
print(next(x))
```

```
<list_iterator object at 0x0000021214C90FD0>
1
2
3
Traceback (most recent call last):
  File "c:\Users\Mao\source\repos\PythonSEP\ppt.py", line 9, in <module>
    print(next(x))
StopIteration
```

Notice how .__next__(x) and next(x) are equivalent.

Iterator Class Example

- Here's a simplified custom range function:

```
class MyRange:
    def __init__(self,n):
        self.i=0
        self.n=n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i<self.n:
            i=self.i
            self.i +=1
            return i
        else:
            raise StopIteration()
```

```
x = MyRange(3)
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
0
1
2
Traceback (most recent call last):
  File "c:\Users\Mao\source\repos\PythonSEP\ppt.py", line 24, in <module>
    print(next(x))
  File "c:\Users\Mao\source\repos\PythonSEP\ppt.py", line 15, in __next__
    raise StopIteration()
StopIteration
```

```
for x in MyRange(3):
    print()
```

```
0
1
2
```

__iter__ and __next__ are what make this class usable as an iterator.

Iterator Class Example

- When an object is passed to the str built-in function, its __str__ method is called.
- When an object is passed to the len built-in function, its __len__ method is called.
- Calling the built-in iter function on an object will attempt to call its __iter__ method.
- Calling the built-in next function on an object will attempt to call its __next__ method.
- The iter function is supposed to return an iterator. So our __iter__ function must return an iterator. But our object ***is*** an iterator, therefore our MyRange object returns self from its __iter__ method because it is its own iterator.

Generators

- The easiest ways to make our own iterators in Python is to create a generator.
- There are two ways to make generators in Python
  - Generator function

```
favorite_numbers = [6, 57, 4, 7, 68, 95]
```

```
def square_all(numbers):
    for n in numbers:
        yield n**2

squares = square_all(favorite_numbers)
```

  - Generator expression

```
squares = (n**2 for n in favorite_numbers)
```

```
>>> type(squares)
<class 'generator'>
>>> next(squares)
36
>>> next(squares)
3249
```

- Both of these objects are of generator type and they're both iterators that provide squares of the numbers in our numbers list.

Generator terminologies

- The word “generator” is used in a few different ways in Python:
  - A **generator**, also called a **generator object**, is an iterator whose type is generator
  - A **generator function** is a special syntax that allows us to make a function which returns a **generator object** when we call it
  - A **generator expression** is a comprehension-like syntax that allows you to create a **generator object** inline

Generator functions

- Generator functions are distinguished from normal functions by the fact that they have one or more yield statements.
- The mere presence of a yield statement turns a function into a generator function.

```
>>> def gimme4_please():
...     print("Let me go get that number for you.")
...     return 4
...
>>> num = gimme4_please()
Let me go get that number for you.
>>> num
4
```

```
>>> def gimme4_later_please():
...     print("Let me go get that number for you.")
...     yield 4
...
>>> get4 = gimme4_later_please()
>>> get4
<generator object gimme4_later_please at 0x7f78b2e7e2b0>
>>> num = next(get4)
Let me go get that number for you.
>>> num
4
```

Generator Function Example

- These two functions have identical behaviors, but the one using yield is much shorter:

```
class Count:

    """Iterator that counts upward forever."""

    def __init__(self, start=0):
        self.num = start

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num += 1
        return num
```

```
def count(start=0):
    num = start
    while True:
        yield num
        num += 1
```

Generator Expression

- Generator expressions(a.k.a.. Generator Comprehension) are a list comprehension-like syntax that allow us to make a generator object.
- They use a shorter inline syntax compared to generator functions, but are not as powerful.
- Only a generator function that can be written in the form on the left can be turned into a generator expression(right).

```
def get_a_generator(some_iterable):
    for item in some_iterable:
        if some_condition(item):
            yield item
```

```
def get_a_generator(some_iterable):
    return (
        item
        for item in some_iterable
        if some_condition(item)
    )
```

Generator Expression Examples

- A list comprehension that filters empty lines from a file and strips newlines from the end can be turned into a generator expression by simply changing the brackets:

```
lines = [
    line.rstrip('\n')
    for line in poem_file
    if line != '\n'
]
```

```
lines = (
    line.rstrip('\n')
    for line in poem_file
    if line != '\n'
)
```

- We can then use either `next()` or `for` to get lines from the generator.

Summary

- To make an iterator, you could create an iterator class, a generator function, or a generator expression.
- Generator expressions
  - Brief and clear
  - Not nearly as flexible as generator functions
  - Recommended when doing simple mapping or filtering
- Generator functions
  - Not as brief, but more powerful than generator expressions
  - Recommended for more complex operations
- Generator class
  - Most amount of work, almost never necessary
  - Useful when extra methods or attributes are needed for your iterator object

File Operation

Antra

- Python provides simple open() and close() for file operations

```
f = open("example.log","r")

for line in f:
  print(line)

f.close()
```

Different Modes to Open Files

Mode	Description
`r`	Opens a file for reading. (default)
`w`	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`x`	Create a new file and open it for writing
`a`	Opens a file for appending at the end of the file. Creates a new file if it does not exist.
`t`	Opens in text mode. (default)
`b`	Opens in binary mode.
`+`	Opens a file for updating (reading and writing)

- Note, when working with text files, it’s recommended to always specify the encoding

```
f = open("example.log","r",encoding="utf-8")
```

with Statement

- Even though we can always use open() and close() to work with files, it is recommended to use the with statement.
- This ensures the file is properly closed when you are finished with it.

```
with open("example.log",'a',encoding = 'utf-8') as f:
    f.write("newline\n")
    f.write("another newline\n\n")
    f.write("last line\n")

with open("example.log",'r',encoding = 'utf-8') as f:
    for i in f:
        print(i)
```

```
2021-02-21 17:25:55,226 - example_logger - DEBUG - This message should go to the log file
2021-02-21 17:25:55,226 - example_logger - INFO - So should this
2021-02-21 17:25:55,227 - example_logger - WARNING - And this, too

newline
another newline

last line
```

File Methods

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the `TextIOBase` and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns `True` if the file stream is interactive.
read(`n`)	Reads at most `n` characters from the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(`n`=-1)	Reads and returns one line from the file. Reads in at most `n` bytes if specified.

File Methods

Method	Description
readlines(`n`=-1)	Reads and returns a list of lines from the file. Reads in at most `n` bytes/characters if specified.
seek(`offset`,`from`=`SEEK_SET`)	Changes the file position to `offset` bytes, in reference to `from` (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(`size`=`None`)	Resizes the file stream to `size` bytes. If `size` is not specified, resizes to current location.
writable()	Returns `True` if the file stream can be written to.
write(`s`)	Writes the string `s` to the file and returns the number of characters written.
writelines(`lines`)	Writes a list of `lines` to the file.

Exception Handling

Antra

Exception Handling in Python

- Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:
  - Exception Handling
  - Assertions

What is an Exception?

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- In general, when a Python application encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python application raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Try-except

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a `try:` block.
- After the `try:` block, include an `except:` statement, followed by a block of code which handles the problem as elegantly as possible.

Try-except Example

- Note that else: will only execute if there are no exceptions

```
try:
    #put your suspicious-looking code here
    apple[0]=orange
except AttributeError: #exception 1
    #if your code run into exception 1
    #this block will be executed
    apple=apple
except IndexError: #exception 2
    #if your code run into exception 2
    #this block will be executed
    orange=orange
except:
    #you can omit the exception in the
    #last execpt to function as a
    #wildcard which catches all exceptions
    orange=apple
    print(banana)
else:
    #the else is optional
    #it must follow all except clauses
    #It is useful for code that must be executed if
    #the try clause does not raise an exception
    banana=banana
    print(banana)
```

Custom Exception

```
class Error(Exception):
    """Base class for other exceptions"""
    pass

class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass

class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
```

```
Enter a number: 12
This value is too large, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

```
# you need to guess this number
number = 10
# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
print("Congratulations! You guessed it correctly.")
```

Custom Exception Class

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Assertion

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- Generally placed before and after logic to check for valid input and output.
- Assert takes a condition/expression and check if it's true
- When assertion fails, AssertionError exception is thrown.

Assertion Example

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32

print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

```
32.0
451
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print KelvinToFahrenheit(-5)
   File "test.py", line 4, in KelvinToFahrenheit
      assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

Logging

Antra

- Python comes with a logging module in the standard library.
- To emit a log message, a caller first requests a named logger.
- The name can be used by the application to configure different rules for different loggers.
- This logger then can be used to emit simply-formatted messages at different log levels (DEBUG, INFO, ERROR, etc.), which again can be used by the application to handle messages of higher priority different than those of a lower priority.

- Basic example

```
import logging
log = logging.getLogger("my-logger")
log.info("Hello, world")
```

- The message is turned into a LogRecord object and routed to a Handler object registered for this logger.
- The handler will then use a Formatter to turn the LogRecord into a string and emit that string.

When to Use Logging?

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	print()
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	logging.info() (or logging.debug() for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	warnings.warn() in library code if the issue is avoidable and the client application should be modified to eliminate the warning logging.warning() if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	logging.error(), logging.exception() or logging.critical() as appropriate for the specific error and application domain

Different Levels of Severity

- By default, only WARNING and above will be tracked

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

Ways to Handle Tracked Events

- Printing to console

```
import logging
logging.warning("Watch out!")
logging.info("I won't get printed")
```

```
WARNING:root:Watch out!
```

Ways to Handle Tracked Events

- Printing to a file

```
import logging

logger = logging.getLogger("example_logger")
logger.addHandler(logging.FileHandler('example.log'))
logger.setLevel(logging.DEBUG)
logger.debug('This message should go to the log file')
logger.info('So should this')
logger.warning('And this, too')
```

example.log

```
1   This message should go to the log file
2   So should this
3   And this, too
4
```

- You can also add logging.StreamHandler() so it's printed to both file and console

```python
import logging

logger = logging.getLogger("example_logger")
#set logging level
logger.setLevel(logging.DEBUG)
#create the handlers -> where to print
file_handler = logging.FileHandler('example.log')
stream_handler = logging.StreamHandler()
#create the printing format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
#set the printing format
file_handler.setFormatter(formatter)
stream_handler.setFormatter(formatter)
#add the handlers
logger.addHandler(file_handler)
logger.addHandler(stream_handler)
logger.debug('This message should go to the log file')
logger.info('So should this')
logger.warning('And this, too')
```

```
2021-02-21 17:25:55,226 - example_logger - DEBUG - This message should go to the log file
2021-02-21 17:25:55,226 - example_logger - INFO - So should this
2021-02-21 17:25:55,227 - example_logger - WARNING - And this, too
```

2021-02-21 17:25:55,226 - example_logger - DEBUG - This message should go to the log file
2021-02-21 17:25:55,226 - example_logger - INFO - So should this
2021-02-21 17:25:55,227 - example_logger - WARNING - And this, too

NumPy

Antra

- NumPy is an open source add-on module for Python that provides common mathematical and numerical routines in pre-compiled, fast functions.
- **NumPy Arrays** are at the core of the package
- Many built-in mathematical functions in NumPy arrays are written in low-level languages such as C or Fortran and **pre-compiled** for fast execution
- NumPy serves as the fundamental building block of more advanced classes such and **pandas** and **DataFrame**

- We can convert a python list into a numpy array

```
import numpy as np

list_1 = [1,2,3]
array_1 = np.array(list_1)

print(type(list_1))
print(type(array_1))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

Adding Lists vs NumPy Arrays

- NumPy arrays are like **mathematical vectors**, adding two vectors is adding the elements of the vectors

```
import numpy as np

list_1 = [1, 2, 3]
array_1 = np.array(list_1)

list_2 = list_1 + list_1
array_2 = array_1 + array_1

print(list_2)
print(array_2)
```

```
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

- In addition to converting a python list into numpy array, we can also create and array directly.
- In python we have range() to create a list on the flight, in numpy, we have arange() which works similarly.

```
print(np.arange(5, 10))  # array of ints 5 to 9
print(np.arange(5, 15, 2))  # array of ints 5 to 15 spaced by 2
print(np.arange(5, 15, 2.5))  # array of decimals 5 to 15 spaced by 2.5
print(np.arange(15, 5, -2.5))  # array of decimals 15 to 5 spaced by -2.5
# linearly spaced numbers
print(np.linspace(0, 10, 4))  # array of 4 decimals spread evenly between 0 and 10 inclusive
```

```
[5 6 7 8 9]
[ 5  7  9 11 13]
[ 5.   7.5 10.  12.5]
[15.  12.5 10.   7.5]
[ 0.          3.33333333  6.66666667 10.        ]
```

- NumPy arrays support many element-wise operations between two arrays

```
import numpy as np

list_1 = [1, 2, 3]
array_1 = np.array(list_1)

array_2 = array_1 * array_1
array_3 = array_1 / array_1
array_4 = array_1 ** array_1

print(array_2)
print(array_3)
print(array_4)
```

```
[1 4 9]
[1. 1. 1.]
[ 1  4 27]
```

Mathematical Operations on an Array

- NumPy has all the built in mathematical functions you can think of
  - output_array = np.method(input_array)
- Look into the documentation for specific usage when needed.

Generating NumPy Arrays and Matrices

- Generating ones/zeroes

```
print(np.zeros((3, 4)), "\n")  # 3x4 matrix of 0
print(np.ones(6), "\n")  # array of 6 ones
print(np.ones((3, 3)) * 5, "\n")  # 3x3 matrix of 5
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[1. 1. 1. 1. 1. 1.]

[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

Generating NumPy Arrays and Matrices

- Generating random numbers

```
from numpy.random import default_rng

# call defualt_rng to get an instance of Generator, which can
# be used to generate random numbers
print(np.random.default_rng().integers(0, 5, (6, 5)), "\n")
print(np.random.default_rng().random((6, 5)))
```

```
[[1 1 0 0 1]
 [3 4 0 0 1]
 [1 3 0 2 0]
 [0 1 0 2 0]
 [1 4 1 2 3]
 [4 3 3 4 2]]

[[0.32734007 0.09619893 0.58627752 0.45437782 0.72661455]
 [0.75178233 0.56398362 0.81268816 0.0655488  0.15483129]
 [0.98865503 0.59440931 0.67278784 0.80687409 0.21934104]
 [0.07375112 0.7998979  0.25351537 0.76445812 0.4022225 ]
 [0.18477093 0.57442218 0.62537233 0.7915803  0.09016944]
 [0.83123823 0.94425729 0.44198018 0.44981708 0.76687723]]
```

Reshaping

```
a = np.random.default_rng().integers(10, 100, 20)
print(a)
print('-' * 60)
b = a.reshape(4, 5)
print(b)
print('-' * 60)
c = a.reshape(2, 2, 5)
print(c)
print('-' * 60)
a = np.random.default_rng().integers(10, 100, 40)
d = a.reshape(2, 2, 2, 5)
print(d)
```

```
[92 37 40 74 16 32 72 95 71 20 46 72 11 51 96 98 96 25 59 19]
------------------------------------------------------------
[[92 37 40 74 16]
 [32 72 95 71 20]
 [46 72 11 51 96]
 [98 96 25 59 19]]
------------------------------------------------------------
[[[92 37 40 74 16]
  [32 72 95 71 20]]

 [[46 72 11 51 96]
  [98 96 25 59 19]]]
------------------------------------------------------------
[[[[23 95 96 39 72]
   [36 24 33 95 17]]

  [[64 24 79 33 63]
   [76 42 78 42 76]]]


 [[[81 24 63 91 78]
   [39 23 32 67 83]]

  [[29 54 98 37 24]
   [12 91 80 11 45]]]]
```

- matrix.revel() will flatten matrix into 1D array

Array Indexing and Slicing

```
mat = np.arange(0, 11)
print(mat)
# everything up to a certain index
print(mat[:6])
# reverse entire array
print(mat[::-1])
# reverse part of array
print(mat[-2:-5:-1])
# reverse, with step
print(mat[-1:-6:-2])
# get elements at multiple indexes
print(mat[[1, 3, -1]])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
[0 1 2 3 4 5]
[10  9  8  7  6  5  4  3  2  1  0]
[9 8 7]
[10  8  6]
[ 1  3 10]
```

Matrix Indexing and Slicing

```python
from numpy.random import default_rng

rng = default_rng()
mat = rng.integers(10, 100, 20).reshape(4, 5)
print(mat)
print('-' * 60)
print(mat[2][3])
print('-' * 60)
print(mat[2, 3])
print('-' * 60)
# get entire row
print(mat[1])
print('-' * 60)
# get entire column
print(mat[:, 2])
print('-' * 60)
# get row 2 to row 4 from column 3
print(mat[1:3, 2])
print('-' * 60)
# get row 1 to row 2 from column 2 to column 3
print(mat[1:3, 2:4])
print('-' * 60)
# get row 1 to row 2 of column 1 and column 3
print(mat[1:3, [1, 3]])
print('-' * 60)
```

```
[[49 60 44 74 69]
 [98 97 90 19 88]
 [60 86 60 89 12]
 [85 48 72 86 49]]
------------------------------------------------------------
89
------------------------------------------------------------
89
------------------------------------------------------------
[98 97 90 19 88]
------------------------------------------------------------
[44 90 60 72]
------------------------------------------------------------
[90 60]
------------------------------------------------------------
[[90 19]
 [60 89]]
------------------------------------------------------------
[[97 19]
 [86 89]]
------------------------------------------------------------
```

Matrix Multiplication

```
mat = np.arange(0, 10).reshape(2, 5)
mat_2 = np.arange(10, 20).reshape(2, 5)
print(mat)
print('-' * 60)
print(mat_2)
print('-' * 60)
# multiply two matrices together
# they need to have the same shape
print(mat * mat_2)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
------------------------------------------------------------
[[10 11 12 13 14]
 [15 16 17 18 19]]
------------------------------------------------------------
[[  0  11  24  39  56]
 [ 75  96 119 144 171]]
```

Matrix Stacking

```
mat_0 = np.zeros((3,3))
mat_1 = np.ones((3,3))
mat_v = np.vstack((mat_0,mat_1))
mat_h = np.hstack((mat_0,mat_1))

print(mat_v)
print("-"*60)
print(mat_h)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
------------------------------------------------------------
[[0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]]
```

pandas

Antra

- Pandas is built on top of NumPy, mainly used to work with relational or labeled data.
- Pandas offers two primary data structures
  - Series -> built on top of NumPy Arrays
  - DataFrame -> built on top of Series

Pandas Series

- You can create a series by providing the value and index separately, you can also create a series from a dictionary
- If index is not provided, default (0,1,2,3...) will be used.

```
import numpy as np
import pandas as pd

my_data = pd.array([1, 2, 3, 4, 5])
my_index = ['a', 'b', 'c', 'd', 'e']

series_1 = pd.Series(data=my_data, index=my_index)
print(series_1)

dic = {'a': 1,
       'b': 2,
       'c': 3,
       'd': 4}
series_2 = pd.Series(dic)
print(series_2)
```

```
a    1
b    2
c    3
d    4
e    5
dtype: Int64
a    1
b    2
c    3
d    4
dtype: int64
```

- In the last slide we are using int as data for simplicity here,but we can store any object in series(including things like dictionaries or even functions)

```
s2 = pd.Series([d1,d2,d3])
print(s2)
s3 = pd.Series([d1.keys(),d1.items(),d1.values()])
print(s3)
s4 = pd.Series([print,len,pow,sum])
print(s4)
```

```
0    {'a': 1, 'b': 2, 'c': 3, 'd': 4}
1    {'a': 1, 'b': 2, 'c': 3, 'd': 4}
2    {'a': 1, 'b': 2, 'c': 3, 'd': 4}
dtype: object
0                          (a, b, c, d)
1    ((a, 1), (b, 2), (c, 3), (d, 4))
2                          (1, 2, 3, 4)
dtype: object
0    <built-in function print>
1      <built-in function len>
2      <built-in function pow>
3      <built-in function sum>
dtype: object
```

Pandas DataFrames

- Similar to a table in relational DB
  - Basically a stack of series under the hood
  - Series are built on top of numpy arrays

Creating a dataframe

- We can create a dataframe from a 2D array
- Default index and columns will be 0,1,2,3….

```
mat = np.random.default_rng().integers(10, 20, 20).reshape(4, 5)
print(mat)
print('-'*40)
row_labels = [0, 1, 2, 3, ]
column_label = ['a', 'b', 'c', 'd', 'e']
df = pd.DataFrame(data=mat, index=row_labels, columns=column_label)
print(df)
```

```
[[16 19 12 14 13]
 [11 17 11 12 15]
 [14 12 15 16 11]
 [10 11 10 13 18]]
----------------------------------------
    a   b   c   d   e
0  16  19  12  14  13
1  11  17  11  12  15
2  14  12  15  16  11
3  10  11  10  13  18
```

- Most of the time, you'll get your dataframe from reading csv/json/text/... files

Peeking a dataframe

- You can use df.head(n)/tail(n) to peek the first/last n(default 5) rows of the df
- you can also use df.info() to get some additional metadata
- df.describe() will provide some descriptive statistics of the df

```
mat = np.random.default_rng().integers(10, 20, 200).reshape(40, 5)
column_label = ['a', 'b', 'c', 'd', 'e']
df = pd.DataFrame(data=mat, columns=column_label)
print(df.info())
print('-'*40)
print(df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 40 entries, 0 to 39
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   a       40 non-null     int64
 1   b       40 non-null     int64
 2   c       40 non-null     int64
 3   d       40 non-null     int64
 4   e       40 non-null     int64
dtypes: int64(5)
memory usage: 1.7 KB
None
----------------------------------------
    a   b   c   d   e
0  19  18  18  11  10
1  17  10  16  18  10
2  18  19  13  12  13
3  12  12  13  17  10
4  17  17  17  10  15
```

```
print(df.describe())
```

	a	b	c	d	e
count	40.000000	40.000000	40.000000	40.000000	40.000000
mean	15.325000	15.925000	14.375000	14.700000	14.875000
std	2.634948	2.421988	3.069181	3.081791	2.775234
min	10.000000	11.000000	10.000000	10.000000	10.000000
25%	13.000000	14.000000	12.000000	12.000000	12.000000
50%	16.000000	16.000000	14.000000	15.000000	15.000000
75%	17.000000	18.000000	17.000000	17.000000	17.000000
max	19.000000	19.000000	19.000000	19.000000	19.000000

Antra

Indexing and slicing a dataframe

- The methods used to index and slice numpy arrays works for pandas columns, except you can now also address columns and rows using their names instead of indexes.
- When the result is a single column, the type will be series, otherwise it will be a new dataframe
- To slice rows, use iloc() if you want to slice with index and loc() to slice with row name

Indexing and slicing a dataframe

```
mat = np.random.default_rng().integers(10, 20, 20).reshape(4, 5)
column_label = ['c0', 'c1', 'c2', 'c3', 'c4']
row_label = ['r0', 'r1', 'r2', 'r3']
df = pd.DataFrame(data=mat, index=row_label, columns=column_label)

print(df.head())
print('-' * 40)

df1 = df[['c1', 'c4']].loc[['r1', 'r3']]
print(df1)
```

```
    c0  c1  c2  c3  c4
r0  13  13  16  19  10
r1  12  17  11  15  13
r2  19  14  10  16  19
r3  11  10  17  19  17
----------------------------------------
    c1  c4
r1  17  13
r3  10  17
```

Creating/Deleting columns/rows

```
mat = np.random.default_rng().integers(10, 20, 20).reshape(4, 5)
column_label = ['c0', 'c1', 'c2', 'c3', 'c4']
row_label = ['r0', 'r1', 'r2', 'r3']
df = pd.DataFrame(data=mat, index=row_label, columns=column_label)

print(df.head())
print('-' * 40)
# adding a new column
df['SUM of c1 and c4'] = df['c1'] + df['c4']
print(df)
print('-' * 40)
# dropping some rows and columns
# the original df is not modified by default
# set inplace=True to modify the original df
df1 = df.drop(['c0', 'c3'], axis=1).drop(['r0', 'r2'], axis=0)
print(df1)
print('-' * 40)
# creating a new df and append to the old df
mat_1 = np.random.default_rng().integers(30, 40, 10).reshape(2, 5)
df_1 = pd.DataFrame(data=mat_1, columns=column_label)
df_2 = df.append(df_1)
print(df_2)
```

```
    c0  c1  c2  c3  c4
r0  19  18  17  12  15
r1  19  17  18  15  10
r2  11  12  13  12  16
r3  17  13  16  18  18
----------------------------------------
    c0  c1  c2  c3  c4  SUM of c1 and c4
r0  19  18  17  12  15                33
r1  19  17  18  15  10                27
r2  11  12  13  12  16                28
r3  17  13  16  18  18                31
----------------------------------------
    c1  c2  c4  SUM of c1 and c4
r1  17  18  10                27
r3  13  16  18                31
----------------------------------------
    c0  c1  c2  c3  c4  SUM of c1 and c4
r0  19  18  17  12  15              33.0
r1  19  17  18  15  10              27.0
r2  11  12  13  12  16              28.0
r3  17  13  16  18  18              31.0
0   38  32  35  30  35               NaN
1   39  33  32  33  31               NaN
```

Demo- Working with SQL Server using Python

- Install **pyodbc**
- Check **https://www.connectionstrings.com** for connection strings used for different databases
- Connect to the DB using pyodbc and conn string
- Create query string, execute on db and get result using **pd.read_sql(query,conn)**
- Insert to DB using cursor