PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2021/2022

IF420 – ANALISIS NUMERIK

Pertemuan ke 2 – OOP in Python

Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 2: Mahasiswa mampu menjelaskan dan menerapkan konsep pemrograman berorientasi obyek di Python – C3

Reviews

- Backgrounds
- Python Basics
- Variables and Basic Data Structures
- Functions
- Branching Statements
- Iteration

Outlines

- Recursion
- Object Oriented Programming (OOP)
- Complexity
- Representation of Numbers
- Errors, Good Programming Practices, and Debugging

Recursion

- Imagine that a CEO of a large company wants to know how many people work for him. One option is to spend a **tremendous amount** of personal effort counting up the number of people on the payroll. However, the CEO has other more important things to do, and so implements another, more **clever, option**. At the next meeting with his department directors, he asks everyone to tell him at the next meeting how many people work for them. Each director then meets with all their managers, who subsequently meet with their supervisors who perform the same task. The supervisors know how many people work under them and readily report this information back to their managers (plus one to count themselves), who relay the aggregated information to the department directors, who relay the relevant information to the CEO.
- This method of solving difficult problems by breaking them up into simpler problems is naturally modeled by **recursive** relationships, and form the basis of important engineering and science problem-solving techniques.

Recursive Function

- A **recursive function** is a function that makes calls to itself. It works like the loops we described before, but sometimes it the situation is better to use recursion than loops.
- Every recursive function has two components: a **base** case and a **recursive** step.
- The **base** case is usually the **smallest input** and has an easily verifiable solution. This is also the mechanism that **stops** the function from calling itself forever. The **recursive** step is the set of **all cases** where a recursive call, or a function call to itself, is made.

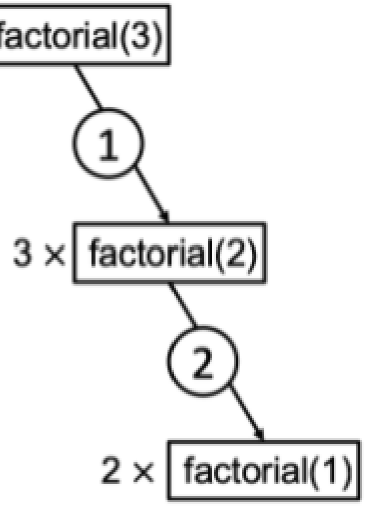
In [1]:
```python
def factorial(n):
    """Computes and returns the factorial of n,
    a positive integer.
    """
    if n == 1: # Base cases!
        return 1
    else: # Recursive step
        return n * factorial(n - 1) # Recursive call
```

In [2]:
```python
factorial(5)
```

Out[2]: 120

Recursive Function

- First recall that when Python executes a function, it creates a workspace for the variables that are created in that function, and whenever a function calls another function, it will wait until that function returns an answer before continuing.
- In programming, this **workspace** is called **stack**. Similar to a stack of plates in our kitchen, elements in a stack are **added** or **removed** from the **top** of the stack to the **bottom**, in a "**last in, first out**" order. For example, in the **np.sin(np.tan(x))**, **sin** must wait for **tan** to return an answer before it can be evaluated. Even though a recursive function makes calls to itself, the same rules apply.
- The order of recursive calls can be depicted by a recursion tree shown in the following figure for factorial(3).
- A **recursion tree** is a diagram of the function calls connected by **numbered arrows** to depict the **order** in which the calls were made.

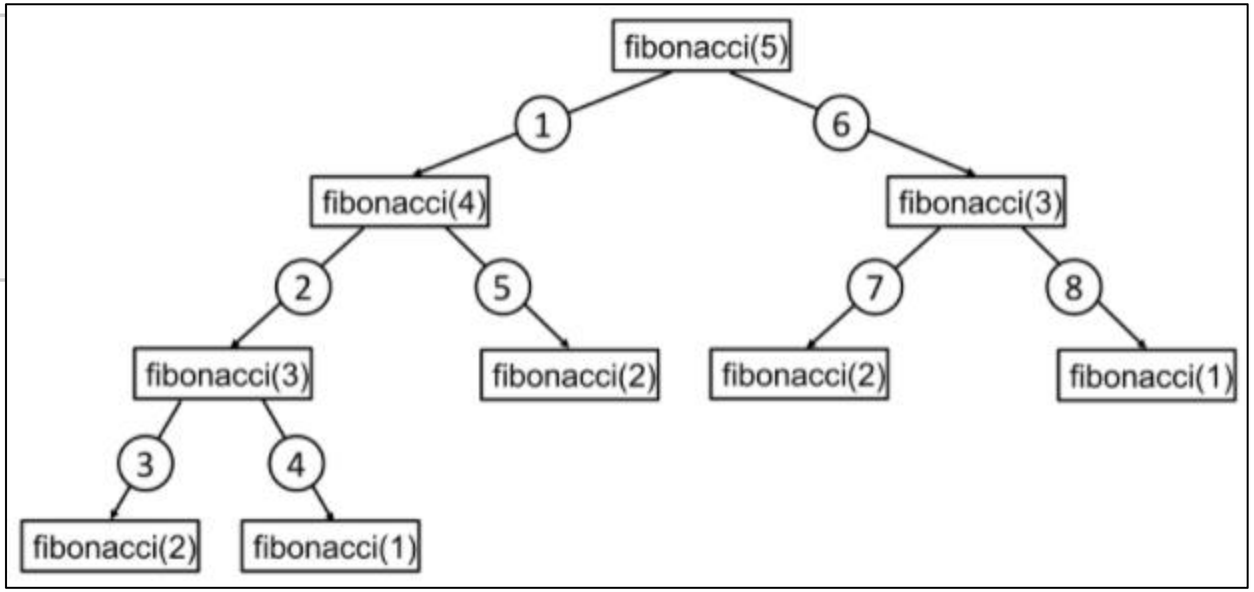


Recursive Function

In [3]:
```python
def fibonacci(n):
    """Computes and returns the Fibonacci of n,
    a postive integer.
    """
    if n == 1: # first base case
        return 1
    elif n == 2: # second base case
        return 1
    else: # Recursive step
        return fibonacci(n-1) + fibonacci(n-2) # Recursive call
```

In [4]:
```python
print(fibonacci(1))
print(fibonacci(2))
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
```

```
1
1
2
3
5
```



```
In [5]: def fibonacci_display(n):
            """Computes and returns the Fibonacci of n,
            a postive integer.
            """
            if n == 1: # first base case
                out = 1
                print(out)
                return out
            elif n == 2: # second base case
                out = 1
                print(out)
                return out
            else: # Recursive step
                out = fibonacci_display(n-1)+fibonacci_display(n-2)
                print(out)
                return out # Recursive call
```

```
In [6]: fibonacci_display(5)

1
1
2
1
3
1
1
2
5

Out[6]: 5
```

- Notice that the number of recursive calls becomes **very large** for even relatively small inputs for n. If you do not agree, try to draw the recursion tree for fibonacci(10).
- If you try your **unmodified** function for inputs around 30, you will notice significant **computation times**.

```
In [7]: fibonacci(30)

Out[7]: 832040
```

Recursive Function

- There is an **iterative** method of computing the n-th Fibonacci number that requires only **one workspace**.

```
In [8]:  import numpy as np

         def iter_fib(n):
             fib = np.ones(n)

             for i in range(2, n):
                 fib[i] = fib[i - 1] + fib[i - 2]

             return fib
```

- Compute the 25-th Fibonacci number using **iter_fib** and **fibonacci**. And use the magic command **timeit** to measure the run time for each. Notice the large difference in running times.

```
In [9]:  %timeit iter_fib(25)

24.4 µs ± 2.09 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [10]: %timeit fibonacci(25)

43.7 ms ± 4.67 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- When we are using **recursive call** as showing above, we need to make sure that it can reach the **base case**, otherwise, it results to **infinite** **recursion**. In Python, when we execute a recursive function on a large output that can not reach the base case, we will encounter a “**maximum recursion depth exceeded error**”.
- We can handle the recursion limit using the **sys** module in Python and set a **higher limit**.

```
In [11]:  factorial(3000)

---------------------------------------------------------------------------
RecursionError                            Traceback (most recent
<ipython-input-11-1f81c569e3b4> in <module>
----> 1 factorial(3000)

<ipython-input-1-f0e38ed114da> in factorial(n)
      6         return 1
      7     else: # Recursive step
----> 8         return n * factorial(n - 1) # Recursive call

... last 1 frames repeated, from the frame below ...

<ipython-input-1-f0e38ed114da> in factorial(n)
      6         return 1
      7     else: # Recursive step
----> 8         return n * factorial(n - 1) # Recursive call

RecursionError: maximum recursion depth exceeded in comparison
```

```
In [12]:  import sys
          sys.setrecursionlimit(10**4)

          factorial(3000)

Out[12]:  4149359603437854085556867093086612170951119194931809917689467657697558 56
```

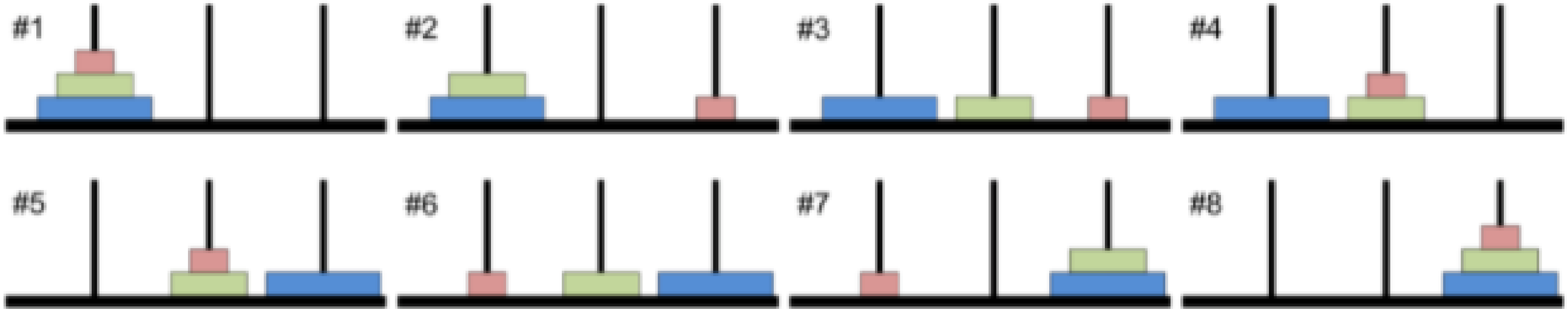
```
41493596034378540855568670930866121709511191949318099176894676576975585 6
```

```
Out[12]:
414935960343785408555686709308661217095111919493180991768946765769755856
```

```
Out[12]: 414935960343785408555686709308661217095111919493180991768946765769755856
         512353195008600076521780034200751846353836171184957508711140459077945534
         021610683396116210379041991775220626633901796828051647196974959688424577
         287660971030037261110953402411271188331577388153284389297376130211063129
         303744014853787254460796102904294910497938881207625116251329170046416689
         621175902035751754889806535778689152850937824699946746991908320935110683
         638242870635222685443392137751504885881040368188090992929124971419005089
         389944047153514731545315874415099601742678750874603679741170723687472771
         439889206836916185036081984597180937844535239585053776110865111623631459
         208861085574508745139453054362137118981508471920944263742032750299963337
         849440147756714146808242074999147148783596697206389546705899601785694802
         633887671128710680049508274007171248194763864013691935443541203127866014
         347925499591435301206531034066255032310207383515021951031486736123387393
         950965514621593490157899499440723110044269248381401414554878727380458560
         235615832043179459530558306933512468907212461514684853087240312679670891
         135489827334753757568993651763964247817334625108790157434373989204922670
         983170339321071763439833524445760404765654004144146994799843545545977993
         867028394285134131889131656953108485135250940061477740470073314065417944
         280044366919036854692708572717016480115120574524486079687737848036606530
```

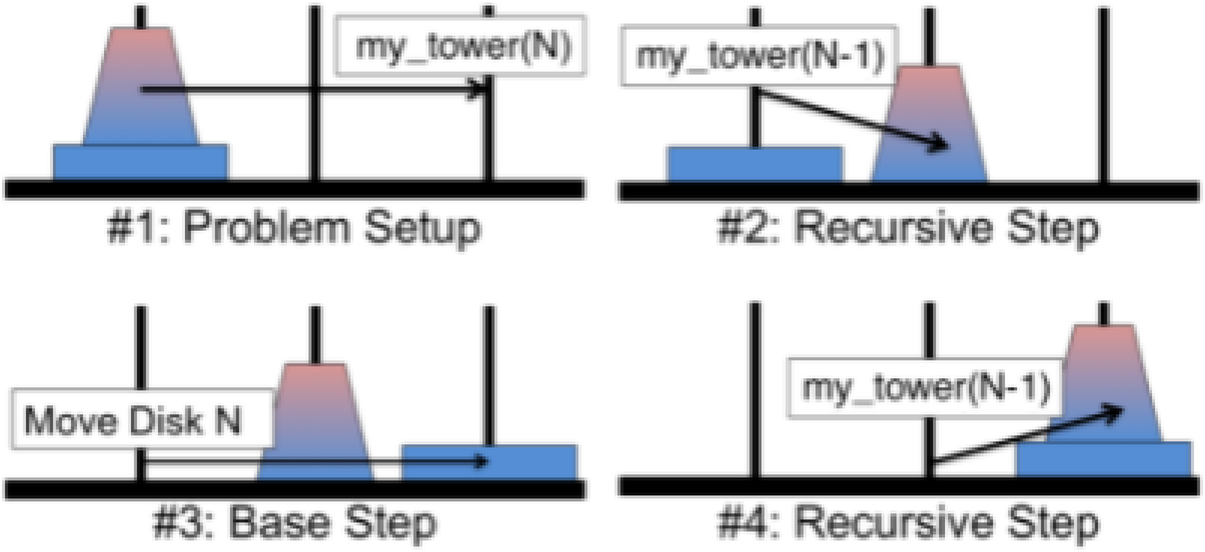
Divide and Conquer

- **Divide and conquer** is a useful strategy for solving difficult problems. Using divide and conquer, difficult problems are solved from **solutions** to many similar **easy problems**. In this way, difficult problems are broken up so they are more **manageable**.
- We cover two classical examples of divide and conquer: the **Towers of Hanoi** Problem and the **Quicksort** algorithm.



Divide and Conquer

- The key to the Towers of Hanoi problem is **breaking it down** into smaller, easier-to-manage problems that we will refer to as **subproblems**. For this problem, it is relatively easy to see that **moving a disk** is easy (which has only **three rules**) but **moving a tower** is difficult (we cannot immediately see how to do it). So we will assign moving a stack of size N to several subproblems of moving a stack of size $N-1$.



In [13]:

```python
def my_towers(N, from_tower, to_tower, alt_tower):
    """
    Displays the moves required to move a tower of size N from the
    'from_tower' to the 'to_tower'.

    'from_tower', 'to_tower' and 'alt_tower' are uniquely either
    1, 2, or 3 referring to tower 1, tower 2, and tower 3.
    """

    if N != 0:
        # recursive call that moves N-1 stack from starting tower
        # to alternate tower
        my_towers(N-1, from_tower, alt_tower, to_tower)

        # display to screen movement of bottom disk from starting
        # tower to final tower
        print("Move disk %d from tower %d to tower %d."\
                  %(N, from_tower, to_tower))

        # recursive call that moves N-1 stack from alternate tower
        # to final tower
        my_towers(N-1, alt_tower, to_tower, from_tower)
```

In [14]:

```python
my_towers(3, 1, 3, 2)
```

```
Move disk 1 from tower 1 to tower 3.
Move disk 2 from tower 1 to tower 2.
Move disk 1 from tower 3 to tower 2.
Move disk 3 from tower 1 to tower 3.
Move disk 1 from tower 2 to tower 1.
Move disk 2 from tower 2 to tower 3.
Move disk 1 from tower 1 to tower 3.
```

Divide and Conquer

- A list of numbers, **A**, is **sorted** if the elements are arranged in **ascending** or **descending** order. Although there are many ways of sorting a list, **quicksort** is a divide-and-conquer approach that is a very fast algorithm for sorting using a single processor (there are faster algorithms for multiple processors).
- The **quicksort algorithm** starts with the observation that sorting a list is hard, but comparison is easy. So instead of sorting a list, we separate the list by comparing to a **pivot**.
- At each recursive call to quicksort, the input list is divided into **three parts**: elements that are **smaller** than the pivot, elements that are **equal** to the pivot, and elements that are **larger** than the pivot. Then a recursive call to quicksort is made on the **two subproblems**: the list of elements **smaller** than the pivot and the list of elements **larger** than the pivot. Eventually the subproblems are **small enough** (i.e., list size of length 1 or 0) that sorting the list is trivial.

In [15]:

```python
def my_quicksort(lst):

    if len(lst) <= 1:
        # list of length 1 is easiest to sort
        # because it is already sorted

        sorted_list = lst
    else:

        # select pivot as the first element of the list
        pivot = lst[0]

        # initialize lists for bigger and smaller elements
        # as well those equal to the pivot
        bigger = []
        smaller = []
        same = []

        # loop through list and put elements into appropriate array

        for item in lst:
            if item > pivot:
                bigger.append(item)
            elif item < pivot:
                smaller.append(item)
            else:
                same.append(item)

        sorted_list = my_quicksort(smaller) + same + my_quicksort(bigger)

    return sorted_list
```

In [16]:

```python
my_quicksort([2, 1, 3, 5, 6, 3, 8, 10])
```

Out[16]:

```
[1, 2, 3, 3, 5, 6, 8, 10]
```

- Similarly to Towers of Hanoi, we have broken up the problem of **sorting** (hard) into many **comparisons** (easy).

Object Oriented Programming (OOP)

- So far, all the codes we have written belong to the category of **procedure-oriented programming** (**POP**), which consists of a list of instructions to tell the computer what to do; these instructions are then organized into **functions**. The program is divided into a collection of **variables**, **data structures**, and **routines** to accomplish different tasks.
- Python is a **multi-paradigm** programming language, which means it supports different programming approach. One different way to program in Python is **object-oriented programming** (**OOP**).
- The object-oriented programming breaks the programming task into **objects**, which combine **data** (known as **attributes**) and **behaviors/functions** (known as **methods**). Therefore, there are two main components of the OOP: **class** and **object**.
- The **class** is a **blueprint** to define a logical grouping of **data** and **functions**. It provides a way to create data structures that model **real-world entities**. While class is the blueprint, an **object** is an **instance** of the class with actual values.

Intro to OOP

In [17]:

```python
class People():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Greetings, " + self.name)
```

In [18]:

```python
person1 = People(name = 'Iron Man', age = 35)
person1.greet()
print(person1.name)
print(person1.age)
```

```
Greetings, Iron Man
Iron Man
35
```

In [19]:

```python
person2 = People(name = 'Batman', age = 33)
person2.greet()
print(person2.name)
print(person2.age)
```

```
Greetings, Batman
Batman
33
```

Intro to OOP

- The **concept** of OOP is to create **reusable** code. There are three key principles of using OOP:

1. **Inheritance** - a way of creating new classes from existing class without modifying it.
2. **Encapsulation** - a way of hiding some of the private details of a class from other objects.
3. **Polymorphism** - a way of using common operation in different ways for different data input.

- With the above principles, there are many **benefits** of using OOP: It provides a clear **modular structure** for programs that enhances code **re-usability**. It provides a simple way to **solve complex** problems. It helps define more **abstract data types** to model real-world scenarios. It **hides** implementation **details**, leaving a clearly defined **interface**. It **combines data** and **operations**.

Class

- A **class** is a definition of the structure that we want. Similar to a **function**, it is defined as a block of code, starting with the **class** statement. The syntax of defining a class is:

```
class ClassName(superclass):

    def __init__(self, arguments):
        # define or assign object attributes

    def other_methods(self, arguments):
        # body of the method
```

- For the class name, it is standard convention to use "**CapWords**." The **superclass** is used when you want create a new class to inherit the attributes and methods from another already defined class.
- The **__init__** is one of the **special methods** in Python classes that is run as soon as an object of a class is instantiated (created). It assigns **initial values** to the object before it is ready to be used.
- The **other_methods** functions are used to define the instance methods that will be applied on the attributes, just like functions we discussed before.

Class

In [20]:

```python
class Student():

    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'

    def say_name(self):
        print("My name is " + self.name)

    def report(self, score):
        self.say_name()
        print("My id is: " + self.sid)
        print("My score is: " + str(score))
```

Object

- An **object** is an **instance** of the defined class with **actual values**.
- We can have many instances of different values associated with the class, and each of these instances will be independent with each other as we saw previously. Also, after we create an object, and call this instance method from the object, we do not need to give value to the self parameter since Python automatically provides it.

In [21]:
```python
student1 = Student("001", "Susan", "F")
student2 = Student("002", "Mike", "M")

student1.say_name()
student2.say_name()
print(student1.type)
print(student1.gender)
```

```
My name is Susan
My name is Mike
learning
F
```

In [22]:
```python
student1.report(95)
student2.report(90)
```

```
My name is Susan
My id is: 001
My score is: 95
My name is Mike
My id is: 002
My score is: 90
```

Class vs Instance Attributes

- The attributes we presented above are actually called **instance attributes**, which means that they are only belong to a specific instance; when you use them, you need to use the **self.attribute** within the class.
- There are another type of attributes called **class attributes**, which will be **shared** with all the instances created from this class.
- In defining a **class attribute**, we must define it **outside** of all the other methods **without** using **self**.
- To use the class attributes, we use **ClassName.attribute**. This attribute will be shared with all the instances that are created from this class.

Class vs Instance Attributes

In [23]:

```python
class Student():

    n_instances = 0

    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'
        Student.n_instances += 1

    def say_name(self):
        print("My name is " + self.name)

    def report(self, score):
        self.say_name()
        print("My id is: " + self.sid)
        print("My score is: " + str(score))

    def num_instances(self):
        print(f'We have {Student.n_instances}-instance in total')
```

In [24]:

```python
student1 = Student("001", "Susan", "F")
student1.num_instances()
student2 = Student("002", "Mike", "M")
student1.num_instances()
student2.num_instances()
```

```
We have 1-instance in total
We have 2-instance in total
We have 2-instance in total
```

Inheritance

- **Inheritance** allows us to define a class that **inherits** all the methods and attributes from another class.
- Convention denotes the new class as **child class**, and the one that it inherits from is called **parent class** or **superclass**.
- If we refer back to the definition of class structure, we can see the structure for basic inheritance is class **ClassName(superclass)**, which means the new class can access all the attributes and methods from the superclass.
- Inheritance builds a **relationship** between the child class and parent class, usually in a way that the parent class is a **general** type while the child class is a **specific** type.

In [25]:

```python
class Sensor():
    def __init__(self, name, location, record_date):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.data = {}

    def add_data(self, t, data):
        self.data['time'] = t
        self.data['data'] = data
        print(f'We have {len(data)} points saved')

    def clear_data(self):
        self.data = {}
        print('Data cleared!')
```

In [26]:

```python
import numpy as np

sensor1 = Sensor('sensor1', 'Berkeley', '2019-01-01')
data = np.random.randint(-10, 10, 10)
sensor1.add_data(np.arange(10), data)
sensor1.data
```

```
We have 10 points saved
```

Out[26]:

```
{'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'data': array([  5, -10,   2,  -8,   2,  -5,  -4,  -5,   3,   8])}
```

Inherit and Extend New Method

- Say we have **one different type** of sensor: an **accelerometer**. It shares the same attributes and methods as **Sensor** class, but it also has different attributes or methods need to be **appended** or **modified** from the original class.
- What should we do? Do we create a different class from scratch?
- This is where inheritance can be used to make life easier. This new class will **inherit** from the **Sensor** class with all the attributes and methods. We can whether we want to **extend** the **attributes** or **methods**.
- Let us first create this new class, **Accelerometer**, and add a new method, **show_type**, to report what kind of sensor it is.

Inherit and Extend New Method

```
In [27]:  class Accelerometer(Sensor):

              def show_type(self):
                  print('I am an accelerometer!')

          acc = Accelerometer('acc1', 'Oakland', '2019-02-01')
          acc.show_type()
          data = np.random.randint(-10, 10, 10)
          acc.add_data(np.arange(10), data)
          acc.data

          I am an accelerometer!
          We have 10 points saved

Out[27]: {'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
          'data': array([ 3,  5, -3, -2, -8, -6,  8, -6, -7,  6])}
```

- This shows the power of inheritance: we have **reused** most part of the Sensor class in a new class, and extended the functionality. Besides, the inheritance sets up a logical relationship for the modeling of the **real-world entities**: the Sensor class as the parent class is more general and passes all the characteristics to the child class Accelerometer.

Inherit and Method Overriding

- When we inherit from a parent class, we can change the implementation of a method provided by the parent class, this is called method **overriding**.

```
In [28]:  class UCBAcc(Accelerometer):

              def show_type(self):
                  print(f'I am {self.name}, created at UC Berkeley!')

          acc_ucb = UCBAcc('UCBAcc', 'Berkeley', '2019-03-01')
          acc_ucb.show_type()

          I am UCBAcc, created at UC Berkeley!
```

- We see that, our new **UCBAcc** class actually overrides the method **show_type** with new features. In this example, we are not only inheriting features from our parent class, but we are also **modifying/improving** some **methods**.

Inherit and Update Attributes with Super

In [29]:
```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.brand = brand
        self.data = {}

new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```
Out[29]: 'XYZ'

- Let us create a class **NewSensor** that inherits from **Sensor** class, but with updated the attributes by adding a new attribute **brand**. Of course, we can re-define the whole **__init__** method as shown below and **overriding** the parent function.

- However, there is a better way to achieve the same. We can use the **super** method to avoid referring to the parent class explicitly.

In [30]:
```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        super().__init__(name, location, record_date)
        self.brand = brand

new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```
Out[30]: 'XYZ'

Encapsulation

- **Encapsulation** is one of the fundamental concepts in OOP. It describes the idea of **restricting access** to **methods** and **attributes** in a class.
- This will hide the complex details from the users, and prevent data being modified by accident. In Python, this is achieved by using **private** methods or attributes using **underscore** as prefix, i.e. single “_” or double “__”.

In [31]:
```python
class Sensor():
    def __init__(self, name, location):
        self.name = name
        self._location = location
        self.__version = '1.0'

    # a getter function
    def get_version(self):
        print(f'The sensor version is {self.__version}')

    # a setter function
    def set_version(self, version):
        self.__version = version
```

```
In [32]: sensor1 = Sensor('Acc', 'Berkeley')
         print(sensor1.name)
         print(sensor1._location)
         print(sensor1.__version)

Acc
Berkeley

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent cal
<ipython-input-32-ca9b481690ba> in <module>
      2 print(sensor1.name)
      3 print(sensor1._location)
----> 4 print(sensor1.__version)

AttributeError: 'Sensor' object has no attribute '__version'
```

```
In [33]: sensor1.get_version()

The sensor version is 1.0

In [34]: sensor1.set_version('2.0')
         sensor1.get_version()

The sensor version is 2.0
```

- The above example shows how the **encapsulation** works.
- With **single underscore**, we defined a **private** variable, and it should not be accessed directly. But this is just convention, nothing stops you from doing that. You can still get access to it if you want to.
- With **double underscore**, we can see that the attribute **__version** can not be accessed or modified directly. Therefore, to get access to the double underscore attributes, we need to use **getter** and **setter** function to access it internally.

Polymorphism

- **Polymorphism** is another fundamental concept in OOP, which means **multiple forms**.
- **Polymorphism** allows us to use a **single interface** with **different underlying forms** such as **data types** or **classes**.
- For example, we can have commonly named **methods** across classes or child classes. We have already seen one example above, when we override the method **show_type** in the **UCBAcc**. For parent class **Accelerometer** and child class **UCBAcc**, they both have a method named **show_type**, but they have different implementation.
- This ability of using single name with many forms acting differently in different situations greatly **reduces** our **complexities**.

Complexity

- Once you have programmed a solution to a problem, an important question is “How long is my program going to run?”
- Clearly the answer to this question depends on many factors, such as the **computer memory**, the computer **speed**, and the **size of the problem**.
- For example, if your function sums every element of a very large array, the time to complete the task will depend on whether your computer can hold the entire array in its memory at once, how fast your computer can do additions, and the size of the array.
- The effort required to run a program to completion is the notion of “**complexity**”.

Complexity

- The complexity of a function is the **relationship** between the size of the **input** and the difficulty of **running** the function to **completion**.
- The size of the input is usually denoted by n. However, n usually describes something more **tangible**, such as the length of an array.
- The difficulty of a problem can be measured in several ways. One suitable way to describe the difficulty of the problem is to use basic operations: **additions**, **subtractions**, **multiplications**, **divisions**, **assignments**, and **function calls**.
- Although each basic operation takes different amounts of time, the number of basic operations needed to complete a function is sufficiently related to the running time to be useful, and it is much easier to count.

Complexity

```python
def f(n):
    out = 0
    for i in range(n):
        for j in range(n):
            out += i*j

    return out
```

- Let’s calculate the number of operations:
- **additions**: n^2, **subtractions**: 0, **multiplications**: n^2, **divisions**: 0, **assignments**: $2n^2 + n + 1$, **function calls**: 0, **total**: $4n^2 + n + 1$.
- The number of assignments is $2n^2 + n + 1$ because the line **out += i∗j** is evaluated n^2 times, **j** is assigned n^2, **i** is assigned n times, and the line **out=0** is assigned once. So, the complexity of the function f can be described as $4n^2 + n + 1$.

Complexity and Big-O Notation

- A common notation for complexity is called **Big-O notation**. Big-O notation establishes the relationship in the **growth** of the number of **basic operations** with respect to the **size** of the **input** as the input size becomes **very large**. Since hardware is different on every machine, we cannot accurately calculate how long it will take to complete without also evaluating the hardware. Then that analysis is only good for that specific machine. We do not really care how long a specific set of input on a specific machine takes. Instead, we will analyze how quickly "**time to completion**" in terms of **basic operations grows** as the **input size grows**, because this analysis is **hardware independent**.
- As n gets large, the **highest power dominates**; therefore, only the **highest power term** is included in **Big-O notation**. Additionally, **coefficients** are not required to characterize growth, and so coefficients are also **dropped**.

Complexity and Big-O Notation

- In the previous example, we counted $4n^2 + n + 1$ basic operations to complete the function. In Big-O notation we would say that the function is $O(n^2)$ (pronounced "O of n-squared").
- We say that any algorithm with complexity $O(n^c)$ where c is some constant with respect to n is **polynomial time**.

```python
def my_fib_iter(n):

    out = [1, 1]

    for i in range(2, n):
        out.append(out[i - 1] + out[i - 2])

    return out
```

- Since the only lines of code that take more time as n grows are those in the for-loop, we can restrict our attention to the for-loop and the code block within it. The code within the **for-loop** does not grow with respect to n (i.e., it is constant). Therefore, the number of basic operations is cn where c is some constant representing the number of basic operations that occur in the for-loop, and these c operations run n times. This gives a complexity of $O(n)$ for **my_fib_iter**.

- Assessing the exact complexity of a function can be difficult. In these cases, it might be sufficient to give an **upper bound** or even an **approximation** of the complexity.

```
def my_fib_rec(n):

    if n < 2:
        out = 1
    else:
        out = my_fib_rec(n-1) + my_fib_rec(n-2)

    return out
```

- As n gets large, we can say that the vast majority of function calls make **two** other **function calls**, **one addition** and **one assignment** to the output. The addition and assignment do not grow with n per function call, so we can ignore them in Big-O notation. However, the number of function calls grows approximately by 2^n, and so the complexity of **my_fib_rec** is upper bound by $O(2^n)$.

- Since the number of recursive calls grows exponentially with n, there is no way the recursive fibonacci function could be polynomial. That is, for any c, there is an n such that **my_fib_rec** takes more than $O(c^n)$ basic operations to complete. Any function that is $O(c^n)$ for some constant c is said to be **exponential time**.

Complexity and Big-O Notation

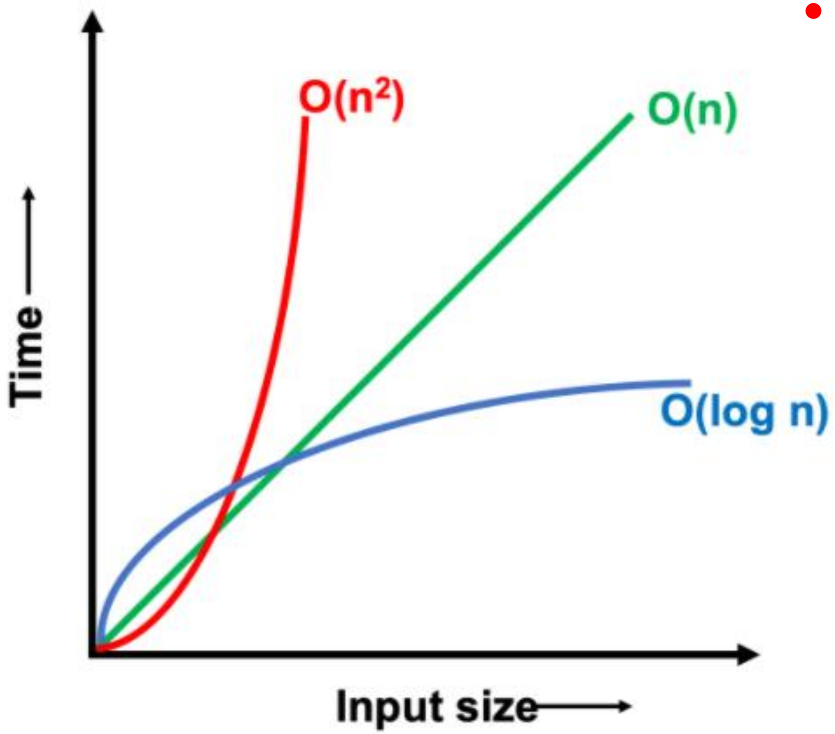
- Again, only the while-loop runs longer for larger n so we can restrict our attention there. Within the while-loop, there are **two assignments**, **one division** and **one addition**, which are both constant time with respect to n. So the complexity depends only on how many times the while-loop runs.

```
def my_divide_by_two(n):

    out = 0
    while n > 1:
        n /= 2
        out += 1

    return out
```

- The while-loop cuts n in half in every iteration until n is less than 1. So the number of iterations, I, is the solution to the equation $\frac{n}{2^I} = 1$. With some manipulation, this solves to $I = \log n$, so the complexity of **my_divide_by_two** is $O(\log n)$. It does not matter what the base of the log is because, recalling log rules, all logs are a scalar multiple of each other.
- Any function with complexity $O(\log n)$ is said to be **log time**.

Complexity Matters

- So why does complexity matter? Because different complexity requires **different time** to complete the task.
- The following figure is a quick sketch showing you how the **time changes** with different input size for complexity $\log(n), n, n^2$.

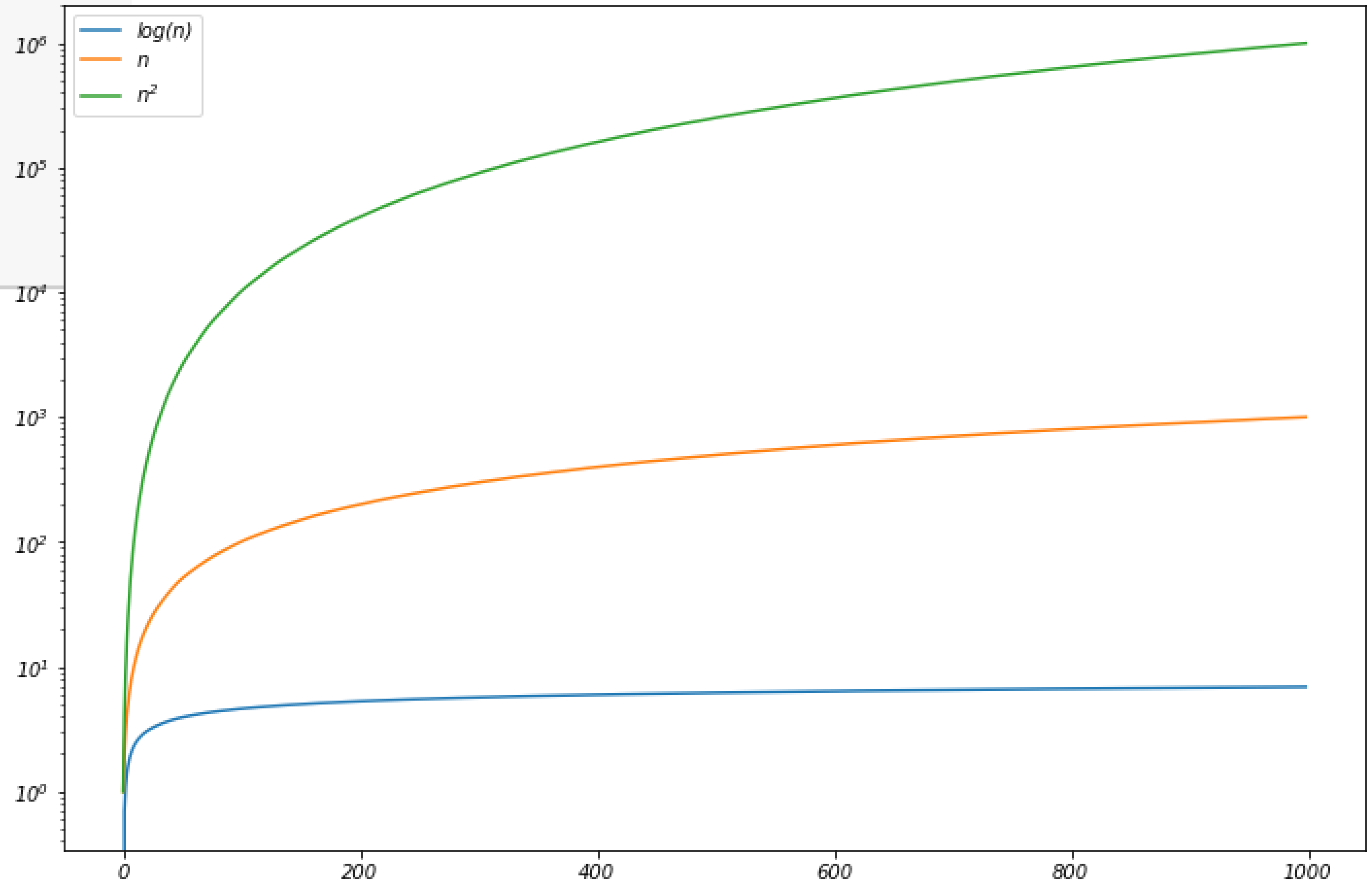


- Let us look at another example. Assume you have an algorithm that runs in exponential time, say $O(2^n)$, and let N be the largest problem you can solve with this algorithm using the computational resources you have, denoted by R. R could be the amount of time you are willing to wait for the function to finish, or R could be the number of basic operations you watch the computer execute before you get sick of waiting. Using the same algorithm, how **large** of a problem can you **solve** given a new computer that is **twice** as fast?

- If we establish $R = 2^N$, using our old computer, with our new computer we have $2R$ computational resources; therefore, we want to find N' such that $2R = 2^{N'}$. With some substitution, we can arrive at $2 \times 2^N = 2^{N'} \rightarrow 2^{N+1} = 2^{N'} \rightarrow N' = N + 1$. So with an **exponential time** algorithm, **doubling** your computational resources will allow you to solve a problem **one unit larger** than you could with your old computer. This is a very small difference. In fact as N gets large, the relative improvement goes to 0.
- With a **polynomial time** algorithm, you can do much better. This time let's assume that $R = N^c$, where c is some constant larger than one. Then $2R = N'^c$, which using similar substitutions as before gets you to $N' = 2^{1/c}N$. So with a **polynomial time** algorithm with power c, you can solve a problem $\sqrt[c]{2}$ **larger** than you could with your old computer. This is a much bigger difference than with the exponential algorithm.
- Finally, let us consider a **log time** algorithm. Let $R = \log N$. Then $2R = \log N'$, and again with some substitution we obtain $N' = N^2$. So with the **double resources**, we can **square the size** of the problem we can solve!

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```
In [3]: plt.figure(figsize = (12, 8))
        n = np.arange(1, 1e3)
        plt.plot(np.log(n), label = 'log(n)')
        plt.plot(n, label = 'n')
        plt.plot(n**2, label = '$n^2$')
        #plt.plot(2**n, label = '$2^n$')
        plt.yscale('log')
        plt.legend()
        plt.show()
```



Magic Command Profiler

- Even if it does not change the **Big-O complexity** of a program, many programmers will spend **long hours** to make their code run **twice** as fast or to gain even small improvements.
- There are ways to check the **run time** of the code in the **Jupyter notebook**, here we will introduce the **magic commands** to do that.
- Notice that the **double percent** magic command will measure the run time for **all the code in a cell**, while the **single percent** command only works for a **single statement**.

```
In [5]:  %time sum(range(200))

Wall time: 0 ns

Out[5]: 19900
```

```
In [6]:  %timeit sum(range(200))

5.47 µs ± 443 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [8]:  %%time
         s = 0
         for i in range(200):
             s += i

Wall time: 0 ns
```

```
In [9]:  %%timeit
         s = 0
         for i in range(200):
             s += i

25.9 µs ± 3.25 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Using Python Profiler

- You could also use the **Python profiler** (you can read more in the Python documentation) to **profile** the code you write.

```
In [10]:  import numpy as np
```

```
In [11]:  def slow_sum(n, m):

              for i in range(n):
                  # we create a size m array of random numbers
                  a = np.random.rand(m)

                  s = 0
                  # in this loop we iterate through the array
                  # and add elements to the sum one by one
                  for j in range(m):
                      s += a[j]
```

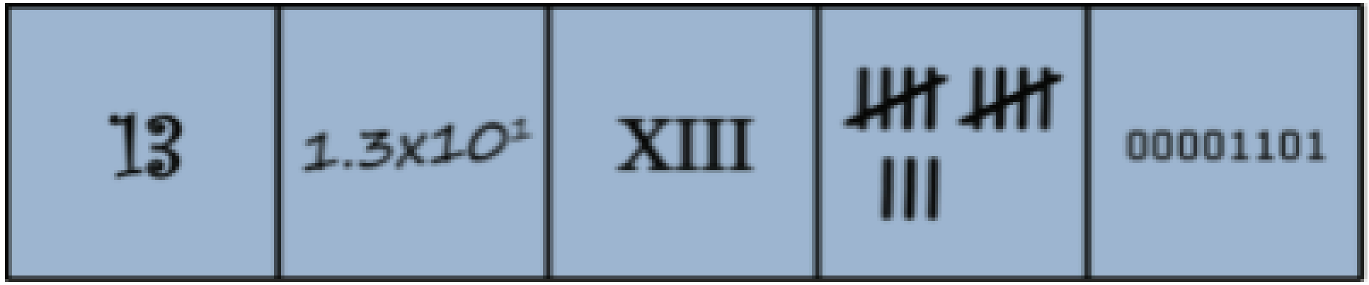
```
In [12]:  %prun slow_sum(1000, 10000)
```

- **ncalls**: for the number of calls,
- **tottime**: for the total time spent in the given function (and excluding time made in calls to sub-functions),
- **percall**: is the quotient of tottime divided by ncalls
- **cumtime** is the total time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- **percall** is the quotient of cumtime divided by primitive calls

```
         1004 function calls in 6.360 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.042    6.042    6.360    6.360 <ipython-input-11-2dd87a83e2f1>:1(slow_sum)
     1000    0.319    0.000    0.319    0.000 {method 'rand' of 'numpy.random.mtrand.RandomState'
objects}
        1    0.000    0.000    6.360    6.360 {built-in method builtins.exec}
        1    0.000    0.000    6.360    6.360 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Representation of Numbers

- There are many ways of representing or writing **numbers**. For example, **decimal** numbers, **Roman** numerals, **scientific notation**, and even **tally marks** are all ways of representing numbers as shown in the following figure.



- Next, we will learn about **different representation** of numbers and how they are useful for computers.
- Besides, we will introduce the **roundoff errors** that associated with the representation of numbers.

Base-N and Binary

- The **decimal** **system** is a way of representing numbers that you are familiar with from elementary school. In the **decimal system**, a number is represented by a list of digits from **0 to 9**, where each digit represents the coefficient for a power of 10.

$$147.3 = 1 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0 + 3 \cdot 10^{-1}.$$

- Since each digit is associated with a power of 10, the decimal system is also known as **base10** because it is based on **10 digits** (0 to 9). However, there is nothing special about base10 numbers except perhaps that you are more accustomed to using them. For example, in **base3** we have the digits 0, 1, and 2 and the number

$$121(base\ 3) = 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 = 9 + 6 + 1 = 16(base\ 10)$$

Base-N and Binary

- A very important representation of numbers for computers is **base2** or **binary** **numbers**. In binary, the only available digits are 0 and 1, and each digit is the coefficient of a power of 2. Digits in a binary number are also known as a bit. Note that binary numbers are still numbers, and so addition and multiplication are defined on them exactly as you learned in grade school.

$$37\ (base\ 10) = 32 + 4 + 1 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 100101\ (base\ 2)$$

$$17\ (base\ 10) = 16 + 1 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 10001\ (base\ 2)$$

Base-N and Binary

$$37 + 17 = 54$$

$$37 \times 17 = 629$$

```
      1
 100101
+ 10001
-------
      0
      1
 100101
+ 10001
-------
     10
      1
 100101
+ 10001
-------
    110
      1
 100101
+ 10001
-------
   0110
      1
 100101
+ 10001
-------
  10110
      1
 100101
+ 10001
-------
 110110  = 32 + 16 + 4 + 2 + 0 = 54 (base10)
```

```
     100101
    x 10001
    -------

     100101
    x 10001
    -------
     100101

     100101
    x 10001
    -------
     100101
          0
         00
        000
+1001010000
-----------
 1001110101 = 512 + 64 + 32 + 16 + 4 + 1 = 629 (base10)
```

- Binary numbers are useful for computers because arithmetic operations on the digits 0 and 1 can be represented using **AND**, **OR**, and **NOT**, which computers can do extremely fast.
- Unlike humans that can abstract numbers to arbitrarily large values, computers have a fixed number of bits that they are capable of storing at one time. For example, a 32-bit computer can represent and process 32-digit binary numbers and no more.

Floating Point Numbers

- The number of bits is usually fixed for any given computer. Using binary representation gives us an **insufficient range** and **precision** of numbers to do relevant engineering calculations. To achieve the range of values needed with the same number of bits, we use **floating point numbers** or **float** for short.
- Instead of utilizing each bit as the coefficient of a power of 2, **floats** allocate bits to three different parts: the **sign** indicator, **s**, which says whether a number is positive or negative; **characteristic** or **exponent**, **e**, which is the power of 2; and the **fraction**, **f**, which is the coefficient of the exponent.
- Almost all platforms map Python floats to the **IEEE754 double precision** - **64** total bits. 1 bit is allocated to the **sign** indicator, 11 bits are allocated to the **exponent**, and 52 bits are allocated to the **fraction**. With 11 bits allocated to the exponent, this makes 2048 values that this number can take.

Floating Point Numbers

- Since we want to be able to make very precise numbers, we want some of these values to represent **negative exponents** (i.e., to allow numbers that are between 0 and 1 (base10)). To accomplish this, 1023 is subtracted from the exponent to normalize it. The value subtracted from the exponent is commonly referred to as the **bias**. The fraction is a number between **1 and 2**. In binary, this means that the leading term will always be 1, and, therefore, it is a waste of bits to store it. To save space, the leading 1 is dropped. In Python, we could get the float information using the **sys** package as shown below:

In [19]:
```
import sys
sys.float_info
```

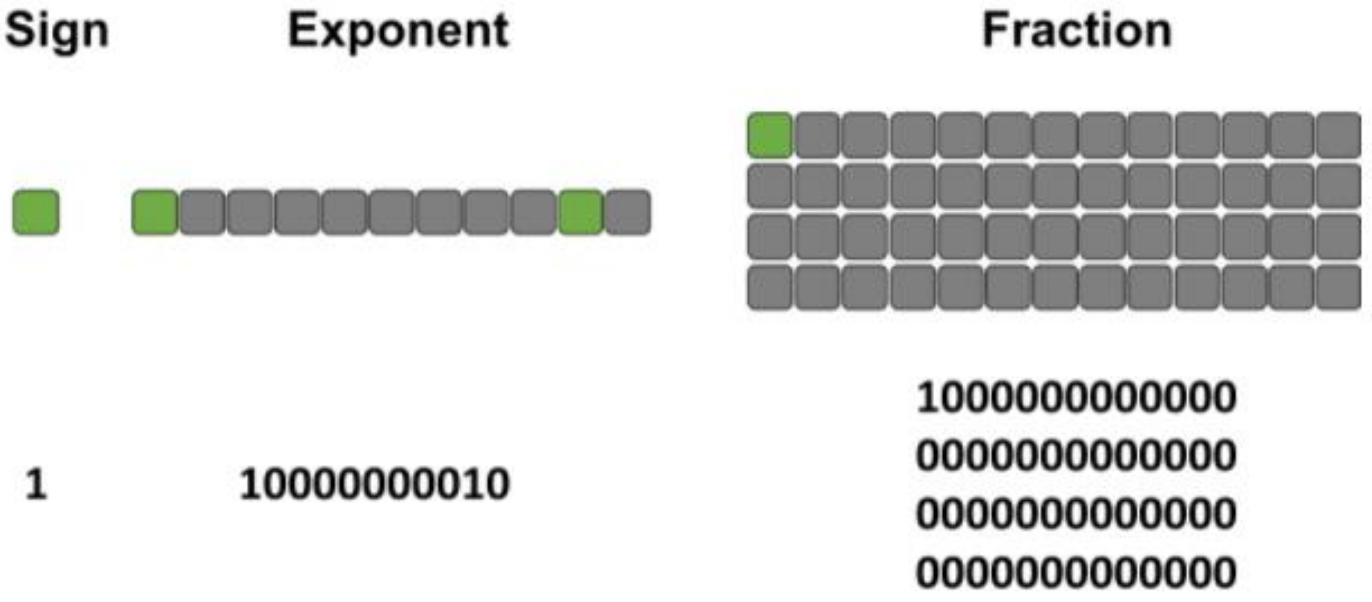
Out[19]:
```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, m
in=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_di
g=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

- A **float** can then be represented as:

$$n = (-1)^s 2^{e-1023}(1+f). \text{ (for 64-bit)}$$

Floating Point Numbers

- What is the number 1 10000000010 1000000000000000000000000000000000000000000000000000 (IEEE754) in base10?
- The exponent in decimal is $1 \cdot 2^{10} + 1 \cdot 2^{1} - 1023 = 3$. The fraction is $1 \cdot \frac{1}{2^1} + 0 \cdot \frac{1}{2^2} + \cdots = 0.5$. Therefore $n = (-1)^1 \cdot 2^3 \cdot (1 + 0.5) = -12.0$ (base10).
- See the following figure for details.



- What is 15.0 (base10) in IEEE754? What is the **largest** number smaller than 15.0? What is the **smallest** number larger than 15.0?
- Since the number is positive, $s = 0$. The largest power of two that is smaller than 15.0 is 8, so the exponent is 3, making the characteristic $3 + 1023 = 1026\ (base10) = 10000000010\ (base2)$. Then the fraction is $\frac{15}{8} - 1 = 0.875\ (base10) = 1 \cdot \frac{1}{2^1} + 1 \cdot \frac{1}{2^2} + 1 \cdot \frac{1}{2^3} = 1110000000000000000000000000000000000000000000000000\ (base2)$. When put together this produces the following conversion: $15\ (base10)\ =$ $0\ 10000000010\ 1110000000000000000000000000000000000000000000000000$
- The **next smallest** number is $0\ 10000000010\ 1101111111111111111111111111111111111111111111111111\ = 14.9999999999999982236431605997$
- The **next largest** number is $0\ 10000000010\ 1110000000000000000000000000000000000000000000000001\ = 15.0000000000000017763568394003$

Floating Point Numbers

- We call the **distance** from one number to the next the **gap**. Because the fraction is multiplied by 2^{e-1023}, the **gap grows** as the **number** represented **grows**. The gap at a given number can be computed using the function **spacing** in **numpy**.
- Example: Use the **spacing** function to determine the gap at 1e9. Verify that adding a number to 1e9 that is less than half the gap at 1e9 results in the same number.

```
In [20]:  import numpy as np

In [21]:  np.spacing(1e9)
Out[21]:  1.1920928955078125e-07

In [22]:  1e9 == (1e9 + np.spacing(1e9)/3)
Out[22]:  True
```

Floating Point Numbers

- There are special cases for the value of a floating point number when e = 0 (i.e., e = 00000000000 (base2)) and when e = 2047 (i.e., e = 11111111111 (base2)), which are reserved.
- When the exponent is 0, the leading 1 in the fraction takes the value 0 instead. The result is a **subnormal number**, which is computed by $n = (-1)^s 2^{-1022}(0 + f)$ (note: it is -1022 instead of -1023).
- When the exponent is 2047 and f is nonzero, then the result is "**Not a Number**", which means that the number is **undefined**.
- When the exponent is 2047, then f = 0 and s = 0, and the result is **positive infinity**.
- When the exponent is 2047, then f = 0, and s = 1, and the result is **minus infinity**.

Floating Point Numbers

```
In [23]:  largest = (2**(2046-1023))*((1 + sum(0.5**np.arange(1, 53))))
          largest

Out[23]:  1.7976931348623157e+308

In [24]:  sys.float_info.max

Out[24]:  1.7976931348623157e+308

In [25]:  smallest = (2**(1-1023))*(1+0)
          smallest

Out[25]:  2.2250738585072014e-308

In [26]:  sys.float_info.min

Out[26]:  2.2250738585072014e-308
```

- Compute the base10 value for 0 11111111110 1111111111111111111111111111111111111111111111111111 (IEEE754), the largest **defined number** for 64 bits, and for 0 00000000001 0000000000000000000000000000000000000000000000000000 (IEEE754), the **smallest**. Note that the exponent is, respectively, e = 2046 and e = 1 to comply with the previously stated rules. Verify that Python agrees with these calculations using **sys.float_info.max** and **sys.float_info.min**.

Floating Point Numbers

- Numbers that are larger than the largest representable floating point number result in **overflow**, and Python handles this case by assigning the result to **inf**. Numbers that are smaller than the smallest subnormal number result in **underflow**, and Python handles this case by assigning the result to **0**.
- **Example**: Show that adding the maximum 64 bits float number with 2 results in the same number. The Python float does not have sufficient precision to store the + 2 for **sys.float_info.max**, therefore, the operations is essentially equivalent to add zero. Also show that adding the maximum 64 bits float number with itself results in **overflow** and that Python assigns this overflow number to **inf**.

```
In [27]:  sys.float_info.max + 2 == sys.float_info.max
Out[27]:  True

In [28]:  sys.float_info.max + sys.float_info.max
Out[28]:  inf
```

- So, what have we gained by using IEEE754 versus binary?
- Using 64 bits binary gives us 2^{64} numbers. Since the number of bits does not change between binary and IEEE754, IEEE754 must also give us 2^{64} numbers.
- In binary, numbers have a **constant spacing** between them. As a result, you cannot have both **range** (i.e., large distance between minimum and maximum representable numbers) and **precision** (i.e., small spacing between numbers). Controlling these parameters would depend on where you put the decimal point in your number. IEEE754 overcomes this limitation by using very **high precision** at **small numbers** and very **low precision** at **large numbers**.
- This limitation is usually **acceptable** because the gap at large numbers is still small relative to the size of the number itself. Therefore, even if the gap is millions large, it is irrelevant to normal calculations if the number under consideration is in the trillions or higher.

Round-off Errors

- In the previous section, we talked about how the floating point numbers are represented in computers as **base 2** fractions. This has a **side effect** that the floating point numbers can not be stored with **perfect precision**, instead the numbers are approximated by finite number of bytes.
- Therefore, the **difference** between an **approximation** of a number used in computation and its **correct** (true) value is called **round-off error**. It is one of the common errors usually in the **numerical calculations**.

Representation of Error

- The most common form of **round-off error** is the **representation error** in the **floating point numbers**. A simple example will be to represent π. We know that π is an **infinite** number, but when we use it, we usually only use a **finite** digits. For example, if you only use 3.14159265, there will be an error between this approximation and the true infinite number. Another example will be $1/3$, the true value will be 0.333333333…, no matter how many decimal digits we choose, there is an round-off error as well.
- Besides, when we **rounding** the numbers **multiple times**, the error will **accumulate**. For instance, if 4.845 is rounded to two decimal places, it is 4.85. Then if we round it again to one decimal place, it is 4.9, the total error will be 0.055. But if we only round one time to one decimal place, it is 4.8, which the error is 0.045.

Round-off Error by Floating-point Arithmetic

- From the previous example, the error between 4.845 and 4.9 should be 0.055. But if you calculate it in Python, you will see the 4.9 - 4.845 is not equal to 0.055.

```
In [36]:  4.9 - 4.845 == 0.055

Out[36]: False
```

- Why does this happen? If we have a look of 4.9 - 4.845, we can see that, we actually get 0.05500000000000604 instead. This is because the floating point can not be represented by the exact number, it is just **approximation**, and when it is used in arithmetic, it is causing a **small error**.

```
In [37]:  4.9 - 4.845

Out[37]: 0.05500000000000604

In [38]:  4.8 - 4.845

Out[38]: -0.04499999999999993
```

Round-off Error by Floating-point Arithmetic

- Another example shows below that 0.1 + 0.2 + 0.3 is not equal 0.6, which has the same cause.

```
In [39]:  0.1 + 0.2 + 0.3 == 0.6

Out[39]: False
```

- Though the numbers cannot be made closer to their intended exact values, the **round** function can be useful for **post-rounding** so that results with inexact values become comparable to one another.

```
In [40]:  round(0.1 + 0.2 + 0.3, 5)  == round(0.6, 5)

Out[40]: True
```

Accumulation of Round-off Error

- When we are doing a sequence of calculations on an initial input with round-off error due to inexact representation, the errors can be **magnified** or **accumulated**.

```
In [41]:  # If we only do once
          1 + 1/3 - 1/3

Out[41]: 1.0
```

```
In [42]:  def add_and_subtract(iterations):
              result = 1

              for i in range(iterations):
                  result += 1/3

              for i in range(iterations):
                  result -= 1/3
              return result
```

```
In [43]:  # If we do this 100 times
          add_and_subtract(100)

Out[43]: 1.0000000000000002
```

```
In [44]:  # If we do this 1000 times
          add_and_subtract(1000)

Out[44]: 1.0000000000000064
```

```
In [45]:  # If we do this 10000 times
          add_and_subtract(10000)

Out[45]: 1.0000000000001166
```

Errors, Good Programming Practices, and Debugging

- Regardless of how proficient, diligent, and careful a programmer you are, writing code with errors is **unavoidable**, and this can be one of the most frustrating parts of programming.
- There are **three basic types** of errors that programmers need to be concerned about: **Syntax errors**, **Runtime errors**, and **Logical errors**.
- **Syntax** is the set of **rules** that govern a language. In written and spoken language, rules can be bent or even broken to accommodate the speaker or writer. However, in a programming language the rules are **rigid**.
- A **syntax error** occurs when the programmer writes an instruction using **incorrect syntax** and Python can **not understand** what you are saying.
- Overall, syntax errors are usually **easily detectable**, **easily found**, and **easily fixed**.

Syntax Error

```
In [46]:  1 = x

  File "<ipython-input-46-7a7b257d8e3d>", line 1
    1 = x
         ^
SyntaxError: can't assign to literal
```

```
In [47]:  (1]

  File "<ipython-input-47-800df0a5e99c>", line 1
    (1]
      ^
SyntaxError: invalid syntax
```

```
In [48]:  if True
              print('Here')

  File "<ipython-input-48-025e9fce1ee3>", line 1
    if True
           ^
SyntaxError: invalid syntax
```

Runtime Error

- Errors that occur during **execution** are called **exceptions** or **runtime errors**. Exceptions are more **difficult** to find and are only **detectable** when a program is **run**.
- **Note**: exceptions are **not fatal**. We will learn later how to handle them in Python. If we do not handle them, Python will **terminate** the program.
- There are different types of **built-in** exceptions: **ZeroDivisionError**, **TypeError**, and **NameError**. You can find a complete list of built-in exceptions in the Python documentation.
- Most of the exceptions are **easy to locate** because Python will **stop running** and tell you where the problem is.
- After programming a function, seasoned programmers will usually run the function several times, allowing the function to “throw” any errors so that they can fix them. But no exception does not mean the function works correctly.

```
In [49]: 1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-49-9e1622b385b6> in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

```
In [50]: x = [2]
         x + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-29a14b9fefb9> in <module>
      1 x = [2]
----> 2 x + 2

TypeError: can only concatenate list (not "int") to list
```

```
In [51]: print(a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-51-bca0e2660b9f> in <module>
----> 1 print(a)

NameError: name 'a' is not defined
```

Logic Error

- One of the most **difficult** kinds of errors to find is called a **logic error**. A logic error does **not throw** an **error** and the program will run smoothly, but is an error because the output you get is **not the solution** you expect.
- For example, consider the following erroneous implementation of the factorial function.

```
In [52]:  def my_bad_factorial(n):
              out = 0
              for i in range(1, n+1):
                  out = out*i

              return out
```

```
In [53]:  my_bad_factorial(4)

Out[53]:  0
```

- This function will not produce a runtime error for any input that is valid for a correctly implemented factorial function. However, if you try using **my_bad_factorial**, you will find that the answer is always 0 because out is initialized to 0 instead of 1.
- When programs become **longer** and more **complicated**, these types of errors are very easy to generate and notoriously difficult to find.

Avoid Errors

1. **Plan** your program

- A good rule of thumb is to plan from the **top to bottom**, and then program from the **bottom to the top**. That is: decide what the overall program is supposed to do, determine what code is necessary to complete the main tasks, and then break the main tasks into components until the module is small enough that you are confident you can write it without errors.

2. **Test** everything often

- You should test often, even within a single module or function. When you are working on a particular module that has several steps, you should perform **intermediate tests** to make sure it is correct up to that point. Then, if you ever get an error, it will probably be in the part of your code written since the last time you did test it.

Avoid Errors

3. Keep your code **clean**

- There are many strategies you can implement to keep your code clean. First, you should write your code in the **fewest instructions** possible.
- You can also keep your code “clean” by using **variables** rather than values.
- You can also keep your code clean by assigning your variables **short, descriptive names**.
- Finally, you can keep your code clean by **commenting** frequently.

```
import numpy as np

s = 0
a = np.random.rand(10)
for i in range(10):
    s = s + a[i]
```

```
n = 10
s = 0
a = np.random.rand(n)

for i in range(n):
    s = s + a[i]
```

```
s = sum(np.random.rand(10))
```

```
y=x**2
y=y+2*x
y=y+1
```

```
y = x**2 + 2*x+1
```

Try/ Except

- A **Try-Except statement** is a code block that allows your program to take **alternative actions** in case an error occurs.

```
try:
    code block 1
except ExceptionName:
    code block 2
```

- If your handler is trying to capture another exception type that the except does not capture it, then we will end up with an **error** and the **execution stops**.

```
In [54]:  x = '6'
          try:
              if x > 3:
                  print('X is larger than 3')
          except TypeError:
              print("Oops! x was not a valid number. Try again...")

          Oops! x was not a valid number. Try again...
```

```
In [55]:  x = '6'
          try:
              if x > 3:
                  print('X is larger than 3')
          except ValueError:
              print("Oops! x was not a valid number. Try again...")

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent ca
<ipython-input-55-899d928e7a1f> in <module>
      1 x = '6'
      2 try:
----> 3     if x > 3:
      4         print('X is larger than 3')
      5 except ValueError:

TypeError: '>' not supported between instances of 'str' and 'int'
```

Try/ Except

- A **try statement** may have **more than one** except statement to handle different exceptions or you can not specify the exception type so that the except will catch any exception.

In [56]:
```python
x = 's'

try:
    if x > 3:
        print(x)
except:
    print(f'Something is wrong with x = {x}')
```
```
Something is wrong with x = s
```

In [57]:
```python
def test_exceptions(x):
    try:
        x = int(x)
        if x > 3:
            print(x)
    except TypeError:
        print("Oops! x was not a valid number. Try again...")
    except ValueError:
        print("Oops! Can not convert x to integer. Try again...")
    except:
        print("Unexpected error")
```

58]:
```python
x = [1, 2]
test_exceptions(x)
```
```
Oops! x was not a valid number. Try again...
```

In [59]:
```python
x = 's'
test_exceptions(x)
```
```
Oops! Can not convert x to integer. Try again...
```

Try/ Except

- Another useful thing in Python is that we can raise some exception in certain cases using **raise**.
- For example, if we need x to be less than or equal to 5, we could use the following code to **raise** an **exception** if x is larger than 5. The program will display our exception and stops the execution.

```
In [60]:  x = 10

          if x > 5:
              raise(Exception('x should be less or equal to 5'))
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-60-99b32b52c4f8> in <module>
      2 
      3 if x > 5:
----> 4     raise(Exception('x should be less or equal to 5'))

Exception: x should be less or equal to 5
```

Type Checking

- Python is both a **strongly** and **dynamically typed** programming language. This means that any variable can take on **any data type** at any time (this is **dynamically typed**), but once a variable is assigned with a type, it **can not change** in unexpected ways.
- In the case of Python, there is no way to ensure that the user of your function is inputting variables of the data type you expect. However, you can have your function type check the input variables before continuing and **force an error** using the **error** function.

In [70]:

```
def my_adder(a, b, c):
    # type check
    if isinstance(a, (float, int, complex)) \
    and isinstance(b, (float, int, complex)) \
    and isinstance(c, (float, int, complex)):
        pass
    else:
        raise(Exception('Input arguments must be numbers'))

    out = a + b + c
    return out
```

```
In [71]: my_adder(1, 2, 3)
Out[71]: 6

In [72]: my_adder(1.0, 2, 3)
Out[72]: 6.0

In [73]: my_adder(1j, 2+2j, 3+2j)
Out[73]: (5+5j)

In [74]: my_adder(1.0, 2.0, '3.0')

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-74-14e4b71b8c1d> in <module>
----> 1 my_adder(1.0, 2.0, '3.0')

<ipython-input-70-a0bf57de5aba> in my_adder(a, b, c)
      6         pass
      7     else:
----> 8         raise(Exception('Input arguments must be numbers'))
      9 
     10     out = a + b + c

Exception: Input arguments must be numbers
```

Debugging

- **Debugging** is the process of systematically **removing errors**, or **bugs**, from your code. Python has functionalities that can assist you when debugging. The standard debugging tool in Python is **pdb** (**Python DeBugger**) for interactive debugging. It lets you step through the code line by line to find out what might be causing a difficult error.
- There are two ways you could debug your code, (1) activate the debugger **after** we run into an exception; (2) activate debugger **before** we run the code.

In [75]:
```python
def square_number(x):

    sq = x**2
    sq += x

    return sq
```

In [76]:
```python
square_number('10')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call la
<ipython-input-76-e0b77a2957d5> in <module>
----> 1 square_number('10')

<ipython-input-75-3fc6a3900214> in square_number(x)
      1 def square_number(x):
      2
----> 3     sq = x**2
      4     sq += x
      5

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

```
In [77]:  %debug

> <ipython-input-75-3fc6a3900214>(3)square_number()
      1 def square_number(x):
      2
----> 3     sq = x**2
      4     sq += x
      5

ipdb> h

Documented commands (type help <topic>):
========================================
EOF    cl         disable  interact  next    psource  rv         unt
a      clear      display  j         p       q        s          until
alias  commands   down     jump      pdef    quit     source     up
args   condition  enable   l         pdoc    r        step       w
b      cont       exit     list      pfile   restart  tbreak     whatis
break  continue   h        ll        pinfo   return   u          where
bt     d          help     longlist  pinfo2  retval   unalias
c      debug      ignore   n         pp      run      undisplay

Miscellaneous help topics:
==========================
exec  pdb

ipdb> p x
'10'
ipdb> type(x)
<class 'str'>
ipdb> p locals()
{'x': '10'}
ipdb> q
```

- After we have this exception, we could activate the debugger by using the magic command - **%debug**, which will open an **interactive debugger** for you.
- You can type in commands in the debugger to get useful information.

Add a Breakpoint

- It is often very useful to insert a breakpoint into your code. A **breakpoint** is a line in your code at which Python will stop when the function is run.

In [81]:
```
import pdb
```

In [82]:
```
def square_number(x):

    sq = x**2

    # we add a breakpoint here
    pdb.set_trace()

    sq += x

    return sq
```

In [83]:
```
square_number(3)
```

```
> <ipython-input-82-e48ec2675aea>(8)square_number()
-> sq += x
(Pdb) l
  3         sq = x**2
  4
  5         # we add a breakpoint here
  6         pdb.set_trace()
  7
  8  ->     sq += x
  9
 10         return sq
[EOF]
(Pdb) p x
3
(Pdb) p sq
9
(Pdb) c
```

Out[83]: 12

Next Week's Outline

- Reading and Writing Data
- Visualization and Plotting
- Parallel Your Python

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Salem, Ariel. 2017. An Easy-To-Use Guide to Big-O Time Complexity. Medium.com. https://medium.com/@ariel.salem1989/an-easy-to-use-guide-to-big-o-time-complexity-5dcf4be8a444
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA UMN UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.