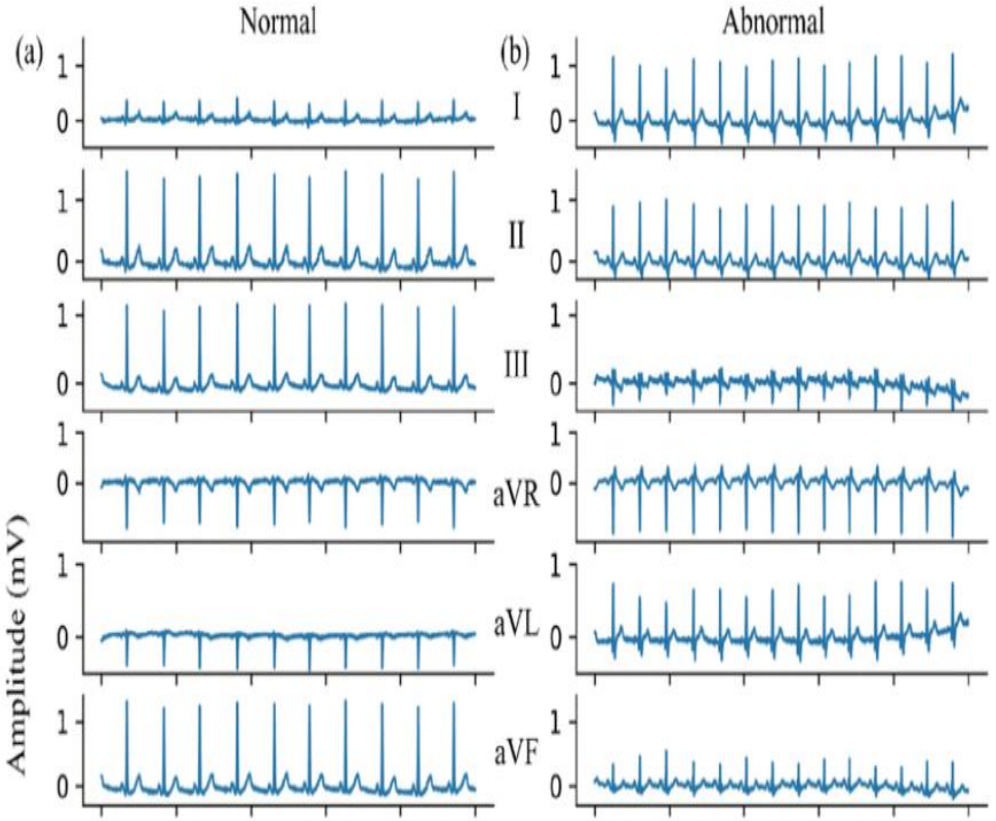
Functions in Python

Introduction

- Why do one requires it?
  - One is having a publicly available dataset ‘A’ of ecg signals having 50,000 time-series instances of ~500 subjects
  - Task is to differentiate between *normal* and *abnormal heart rate* (known as *arrhythmia*)
  - How to get it done ?
    - *“**Preprocess**”* data to normalize and remove noise (occurring due to various factors such as device errors, subject state, human errors etc.) in the data set
    - “***Deploy***” already existing/novel algorithm to classify any signal as *normal* or *abnormal*
    - *“**Evaluate**”* the performance of the above algorithm on other dataset ‘B’ not having any signal of dataset ‘A’
  - Will it make sense if all the three steps mentioned above are put in a single sequence of instructions?



Source: [(a) Normal ECG sample and (b) abnormal ECG sample in preliminary dataset. | Download Scientific Diagram (researchgate.net)]

Introduction

- It makes sense to ***separate the tasks*** into clearly defined ***subtasks*** to conveniently and efficiently execute the whole process

- This is known as ***"Modularization"*** of the process/program, which requires the concept of "*Function*"

- Any function takes a set of '*input' parameters* and optionally return a set of '*output*' parameters

  - Example: len(string name) will take one input parameter of type string and will return the number of characters it contains

Introduction

- ***Function*** is a ***sequence of instructions*** that performs a specific task.
- Help programmers to structure their code and make it easier to read.
- The 'def' keyword, the function name, and any parameters included in parenthesis define a function.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Types of Functions in Python

- Two types of functions:
  - ➢ In-built functions
    - ❖ Provide (hopefully) efficient body of instructions in already existing packages of Python to efficiently carry out sub-tasks of (complex) tasks
    - ❖ One first import the corresponding package and then call the function using that package
      - ❑ Example:

```
In [63]: import numpy as np

In [75]: z = np.arange(2,501,2)
         z

Out[75]: array([  2,   4,   6,   8,  10,  12,  14,  16,  18,  20,  22,  24,  26,
                 28,  30,  32,  34,  36,  38,  40,  42,  44,  46,  48,  50,  52,
                 54,  56,  58,  60,  62,  64,  66,  68,  70,  72,  74,  76,  78,
                 80,  82,  84,  86,  88,  90,  92,  94,  96,  98, 100, 102, 104,
                106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130,
                132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156,
                158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182,
                184, 186, 188, 190, 192, 194, 196, 198, 200, 202, 204, 206, 208,
                210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234,
                236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 256, 258, 260,
                262, 264, 266, 268, 270, 272, 274, 276, 278, 280, 282, 284, 286,
                288, 290, 292, 294, 296, 298, 300, 302, 304, 306, 308, 310, 312,
                314, 316, 318, 320, 322, 324, 326, 328, 330, 332, 334, 336, 338,
                340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 360, 362, 364,
                366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390,
                392, 394, 396, 398, 400, 402, 404, 406, 408, 410, 412, 414, 416,
                418, 420, 422, 424, 426, 428, 430, 432, 434, 436, 438, 440, 442,
                444, 446, 448, 450, 452, 454, 456, 458, 460, 462, 464, 466, 468,
                470, 472, 474, 476, 478, 480, 482, 484, 486, 488, 490, 492, 494,
                496, 498, 500])
```

Function: Header, Body and Return

- User defined functions
  - Define your own function using keyword '*def*'

```
def function_name(input param_1, input param_2, …):
        # Comments about the function

        statement 1    # Comment about the statement 1
        statement 2    # Comment about the statement 2
                ⋮
        statement n    # Comment about the statement n
        return output_parameters     # Optional

        # End of the body of function
```

Creating and Calling a Function

```
def my_function():
  print("Hello from a function")

my_function()
```

```
def add_numbers(num1, num2):
  result = num1 + num2
  return result

x = add_numbers(7, 3)
```

Fair Practices for Function

- *Parameter*: Variable defined by the function which receives a value when the function is invoked
- *Argument*: Value passed to a function when it is invoked

- Example:
  - For a function: *hello*(name), name is the input parameter. When we call the function and pass the value ‘Steve’ then this value is the input argument
- Due to *very subtle difference* these two terms are often used interchangeably

- It is ***customary*** to put a description of the function, author, and creation date in the descriptive string under the function header using single line or multi-line comments

Fair Practices for Function…

- Functions must conform to a naming scheme similar to variables

  - ➢ They can only contain *alphanumeric characters* and *underscores*
  - ➢ The first character must be a letter
    - ❖ Avoid using _ as the first character in a function/variable name
  - ➢ It is a good programming practice to provide the function name in lower case with words separated by underscore to improve it readability

  Example:

In [1]:
```
def my_mult(a,b,c):
    """This function returns product of three numbers
        Author:
        Date:
    """
    output = a*b*c
    return output
```

In [4]:
```
a = 7; b = 9
c = 3
d = my_mult(a,b,c)
print(d)
```

```
189
```

Function having Multiple Output Parameters

- Python functions can have multiple output parameters
  - ➢ Unpack the results with multiple variables separated by commas
  - ➢ If one uses only one variable then one will get a tuple containing all the output parameters

In [22]:
```
def my_trig_sum(a, b):
    """
    function to demo return multiple
    author
    date
    """
    out1 = np.sin(a) + np.cos(b)
    out2 = np.sin(b) + np.cos(a)
    return out1, out2, [out1, out2]

c,d,e = my_trig_sum(np.pi/6.0, np.pi)
print(f"c = {c}, d = {d}, e = {e}")
```

```
c = -0.5, d = 0.8660254037844388, e = [-0.5, 0.8660254037844388]
```

Function having No Input and Output

- Input and Output Parameters in any user defined function are optional:

```
In [23]: def print_hello():
             print("Hello")

         print_hello()
```

```
Hello
```

- Python provides the facility of *"default value"* for the input of the argument as well:

```
In [25]: def print_greeting(day = "Monday", name = "Patrick"):
             print(f"Greetings, {name}, today is {day}")

         print_greeting()
         print_greeting('Tuesday', 'Bob')
```

```
Greetings, Patrick, today is Monday
Greetings, Bob, today is Tuesday
```

Default Parameter Value

```
def my_function(university_name = “SNIoE”):
  print(“I am studying at " + university_name)
```

my_function(“IIT Kanpur") ➔ Output : “I am studying at IIT Kanpur "

my_function (“IISc Bengaluru”) ➔ Output : “I am studying at IISc Bengaluru"

my_function() ➔ Output : “I am studying at SNIoE"

Arguments in a Function

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses.
- You can add as many arguments as you want, just separate them with a comma.
- The following example is a function with two arguments (num1 and num2). When the function is called, we pass two numbers which is used inside the function to get the result

```
def add_numbers(num1, num2):
  result = num1 + num2
  return result
```

Number of Arguments

- A function must be called with the correct number of arguments.

- If your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)
```

my_function(“Sumit", “Shekhar") **: Correct**

my_function (“Sumit", “Shekhar“, “Sharma”) **: Error**

my_function(“Sumit") **: Error**

Type of Arguments

- Similarly define *my_add*(a,b,c) function:

```
In [12]: def my_add(a,b,c):
             """This function returns addition of three numbers
                 Author:
                 Date:
             """
             output = a + b + c
             return output
```

- What will be the output: ?

```
In [13]: a= 2
         b = 2
         d = my_add(a,b,'3')
         print(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 3
      1 a= 2
      2 b = 2
----> 3 d = my_add(a,b,'3')
      4 print(d)

Cell In[12], line 6, in my_add(a, b, c)
      1 def my_add(a,b,c):
      2     """This function returns product of three numbers
      3         Author:
      4         Date:
      5     """
----> 6     output = a + b + c
      7     return output

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Local and Global Variables

- Some more alternatives:

```
In [26]: print_greeting(name = 'Bob')
         Greetings, Bob, today is Monday
```

- A memory block is associated with ***notebook environment*** and all the variables created in the notebook are stored (globally)

- Similarly every function owns a memory block which is not shared with other memory blocks including the memory block of notebook environment

  - So a variable with name '*x*' created in the memory block can be modified in the function without changing the state of the variable with the same name '*x*' outside the function

Local and Global Variables

- Example:

In [28]:
```
def my_mult(a, b, c):
    out = a*b*c
    print(f"The value out within the function is {out}")
    return out

out = 17
d = my_mult(1, 2, 3)
print(f"The value out outside the function is {out}")
```

```
The value out within the function is 6
The value out outside the function is 17
```

- What will happen in the following case:

In [ ]:
```
n = 42
def func():
    print(f"Within function: n is {n}")
    n = 3
    print(f"Within function: change n to {n}")
func()
print(f"Outside function: Value of n is {n}")
```

Local and Global Variables

- It will show error as:

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[2], line 6
      4     n = 3
      5     print(f"Within function: change n to {n}")
----> 6 func()
      7 print(f"Outside function: Value of n is {n}")

Cell In[2], line 3, in func()
      2 def func():
----> 3     print(f"Within function: n is {n}")
      4     n = 3
      5     print(f"Within function: change n to {n}")

UnboundLocalError: local variable 'n' referenced before assignment
```

- Why it is producing error?
  - There will exist separate global (jupyter notebook environment) and local (function specific) level memory blocks
  - Interpreter will parse the expressions and statements before execution
  - While parsing it finds out that there are two variables with the same name in global and local memory blocks
  - Subsequently to resolve the conflict, it will give priority to the local variable (here n) inside the corresponding function
    - Which will result in error at line 3

Local and Global Variables

- How to resolve this error?
  - ➢ Depends on what we are trying to achieve using the code

    - ❖ If one is required to define two variables with the same name at the global and local levels then, at the local level it will be the local variable which will be identified and later on processed by the interpreter.
      - ❑ So, don't use the variable before defining it

    - ❖ Otherwise, If one is required to use only global variable inside the function, then interpreter has to be informed about it (as it will by default consider the definition $n = 42$ outside the function and definition $n = 3$ inside the function corresponding to two different variables)
      - ❑ To resolve this, one is required to use keyword "***global***" provided by python

Local and Global Variables

- Now:

In [4]:
```
n = 42
def func():
    global n
    print(f"Within function: n is {n}")
    n = 3
    print(f"Within function: change n to {n}")
func()
print(f"Outside function: Value of n is {n}")
```

```
Within function: n is 42
Within function: change n to 3
```

- What will be the output of the last line ?

Lambda Function

- Some time it might not be the optimal way to define a function using keyword "*def*" and subsequently writing the statements and expressions

  - Specifically, when the function code size is very small and it is not used (called) frequently; which may lead to slow the overall execution time

  - Here, one may use anonymous function (function without name) using "lambda" keyword as:
    lambda arguments : expression

Examples of *Lambda* Function

- Example:

```
In [18]: square = lambda x: x**2
         print(square(2))
         print(square(5))
```

```
4
25
```

- What will happen in the following case:

```
In [29]: add_var = lambda x,y: x + y
         print(add_var(2,7))
         print(add_var('Hello','World'))
```

```
9
HelloWorld
```

Recursive Function

- Recursive function is a function which **makes a call to itself**
- Example: $f(n) = \begin{cases} 1, & n = 0 \\ n * f(n-1), & otherwise \end{cases}$
  - Base-case: $\mathrm{n} = 0, f(0) = 1$
  - Recursive-step: $\mathrm{n} > 0, f(n) = n * f(n-1)$

In [19]:
```
def factorial(n):
    """Computes and returns the factorial of n,
    a positive integer.
    """
    if n == 0: # Base case
        return 1
    return n*factorial(n-1)

factorial(6)
```

Out[19]: 720

Another Example: Fibonacci Sequence

- Consider a sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 \ldots$
- What does one observe here ?
  - ***n^{th} term*** in the above sequence be $f(n) \Rightarrow f(n) = f(n-1) + f(n-2)$
  - How to write it in terms of ***basic and recursive steps*** ?

$$f(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

In [5]:
```
def fibonacci(n):
    if (n == 1 or n == 2):
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

for i in range(1,15):
    print(fibonacci(i))
```

Iterative Version of Fibonacci Sequence

- Iterative version:

In [7]:
```
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

print(iter_fib(14))
```

```
[  1.   1.   2.   3.   5.   8.  13.  21.  34.  55.  89. 144. 233. 377.]
```

- How long does it take to compute the iterative and recursive version?

In [6]:
```
%timeit iter_fib(25)
```

```
6.04 µs ± 121 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [8]:
```
%timeit fibonacci(25)
```

```
9.99 ms ± 213 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Golden Ratio

- Can you compute the ratio $\mathrm{G(n)} = \frac{fib(n)}{fib(n-1)}$, $for\ n = 2, 3, 4, 5, 6, 7, \ldots$?
  - $\frac{f(2)}{f(1)} = 1,\ \frac{f(3)}{f(2)} = 2,\ \frac{f(4)}{f(3)} = 1.5,\ \frac{f(5)}{f(4)} = 1.66,\ \frac{f(6)}{f(5)} = 1.6,\ \frac{f(7)}{f(6)} = 1.625, \frac{f(8)}{f(7)} = 1.615$
  - $\frac{f(9)}{f(8)} = 1.619,\ \frac{f(10)}{f(9)} = 1.618,\ \frac{f(11)}{f(10)} = 1.618,\ \frac{f(12)}{f(11)} = 1.618,\ \frac{f(13)}{f(12)} = 1.618,\ \frac{f(14)}{f(13)} = 1.618$

Golden Ratio

- The golden ratio, ϕ, is the limit of $f(n+1)/f(n)$ as n goes to infinity where $f(n)$ is the n^{th} Fibonacci number, which can be shown to be exactly $\frac{1+\sqrt{5}}{2}$ and is approximately 1.62. We say that $G(n) = f(n+1)/f(n)$ is the n^{th} approximation of the golden ratio and $G(1) = 1$. It can be shown that golden ratio ϕ is also the limit of the continued fraction: $\phi = 1 + \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}$.

- Write a recursive function with the header $my\_golden\_ratio(n)$, where the output is the n^{th} approximation of the golden ratio according to the continued fraction recursive relationship. Use the ***continued fraction approximation*** for the golden ratio and not the $G(n) = f(n+1)/f(n)$ definition; however, for both definitions, $G(1) = 1$. Studies have shown that rectangles with aspect ratio (i.e., length divided by width) close to the golden ratio are more pleasing to the eye than rectangles that are not. What is the aspect ratio of many wide-screen TVs and movie screens?

Golden Ratio

In [12]:
```
import sys
def my_golden_ratio(n):
    """Computes and returns the factorial of n,
    a positive integer.
    """
    if n == 1: # Base case
        return 1
    return 1 + (1/my_golden_ratio(n-1))

my_golden_ratio(2000)
```

Out[12]:
```
1.618033988749895
```