1. Reutilización

Uno de los aspectos más importantes en el mundo de la programación es el de la reutilización. Consiste en utilizar algoritmos ya codificados (muchas veces ya compilados), que están *bien probados* y simplifican la producción de nuevo software, re-aprovechando lo que existe. Dentro de estos casos, en `Python` ya proporciona multitudes de estas porciones de código. Esto lo haremos a través de *funciones* que ya están a nuestra disposición.

Las funciones suelen estar agrupadas dentro de *bibliotecas* (*libraries* o simplemente *libs*), las cuales se encuentran agrupadas por su propósito o funcionalidad. Dentro de éstas podemos encontrar funciones matemáticas, como potenciación, redondeo, valor absoluto, seno, tangente, etc. Así como también pueden contener **valores constantes**, por ejemplo `math.pi` que contiene el valor de π.

Para utilizar alguna de estas bibliotecas, debemos agregar el módulo, para ello se recurre al comando `import`, seguido del nombre . Esto usualmente se encuentra al principio del código fuente, en las primeras lineas, pudiendo agregar sólo una a la vez y por linea:

```
import module1
import module2
# ...
import moduleX
```

1.1. Invocar una función

Las funciones son invocadas o llamadas desde alguna parte de nuestro algoritmo para cumplimentar una tarea. Un ejemplo de llamado a función y su sintaxis sería el siguiente:

Programa 1: Distancia entre dos números

```
def main():
  print("Ingrese dos números reales: ")
  n1 = float(input())
  n2 = float(input())

  # Llamado a la función incluida abs() "valor absoluto"
  d = abs(n2-n1)
  print(f"La distancia entre ambos es: {d}")

main()
```

Como se observa en el Programa 1, para calcular la distancia entre dos números, se utiliza la función `abs()` la cual **recibe** un número como argumento (en este caso el resultado de `n2-n1`) y **retorna** o *devuelve* un valor que se almacena en la variable `d`.

De hecho se han estado utilizando varias funciones del *módulo principal* (incluido por defecto) de `Python` como son `print()`, `input()`, `type()`, etc. Estas funciones están a nuestra disposición sin la necesidad de utilizar un `import`.

Observemos el siguiente programa que utiliza un `import` necesario para acceder a una función matemática:

Programa 2: Potenciación

```python
import math

def main():
  print("Ingrese dos números: ")
  n1 = float(input())
  n2 = float(input())

  pot = math.pow(n1, n2) # n1^n2
  print(f"{n1} elevado a la {n2} es: {pot}")

main()
```

Una posible salida de este programa podría ser:

```
Ingrese dos números:
2
3
2.0 elevado a la 3.0 es: 8.0
```

En esta oportunidad vemos que la función *potenciación* (`pow`) posee dos parámetros, los cuales se utilizan como base y exponente dentro del algoritmo que ejecuta.

2. Funciones y constantes matemáticas

Muchas de las actividades y ejercitaciones propuestas requieren de la utilización de procesos matemáticos como se ha expuesto en el apartado anterior con la distancia entre dos números reales. El uso de algoritmos donde están involucradas las matemáticas son innumerables, es por ello que ya hay muchos algoritmos de uso común dentro de la biblioteca estándar matemática a la cual accedemos a través de `math`.

A continuación se emplearán algunas de estas funciones que se tienen a disposición.

2.1. Valores absolutos, Techo, Piso, Truncamiento y Redondeo

2.1.1. Funciones

```python
# Valor Absoluto: Retorna el valor númerico siempre positivo.
math.fabs(x) # siempre devuelve un float
# Techo: retorna el valor redondeo al entero superior.
math.ceil(x)
# Piso: retorna el valor redondeo al entero inferior.
math.floor(x)
# Redondeo: retorna el valor redondeo al entero más próximo.
round(x) # incluida en el módulo principal (no en math)
# Truncamiento: retorna el valor entero.
math.trunc(x)
```

2.1.2. Ejemplos

```python
import math # para utilizar fabs, floor, etc.

def main():
  print("VALOR ABSOLUTO")
  print(f"abs(3.77) -> {math.fabs(3.77)}")
  print(f"abs(-2.33) -> {math.fabs(-2.33)}\n")

  print("REDONDEO")
  print(f"round(3.77) -> {round(3.77)}")
  print(f"round(-2.33) -> {round(-2.33)}\n")

  print("PISO")
  print(f"floor(3.77) -> {math.floor(3.77)}")
  print(f"floor(-2.33) -> {math.floor(-2.33)}\n")

  print("TECHO\n")
  print(f"ciel(3.77) -> {math.ceil(3.77)}")
  print(f"ciel(-2.33) -> {math.ceil(-2.33)}\n")

  print("TRUNCAMIENTO\n")
  print(f"trunc(3.77) -> {math.trunc(3.77)}")
  print(f"trunc(-2.33) -> {math.trunc(-2.33)}\n")

main()
```

2.2. Funciones trigonométricas

2.3. Funciones

Las funciones trigonométricas reciben un parámetro cuyo valor debe estar expresado en radianes.

```python
math.acos(x)  # Arco-coseno
math.asin(x)  # Arco-seno
math.atan(x)  # Arco-tangente
math.atan2(x) # Arco-tangente cuadrada
math.cos(x)   # Coseno
math.cosh(x)  # Coseno hiperbólico
math.sin(x)   # Seno
math.sinh(x)  # Seno hiperbólico
math.tan(x)   # Tangente
math.tanh(x)  # Tangente hiperbólica
```

2.3.1. Ejemplos

```python
import math # sin, cos, tan, asin, acos, atan

def main():
  print("Trigonométricas básicas: ")
  print(f"sin(pi/2) -> {math.sin(math.pi/2)}")
  print(f"con(pi)   -> {math.cos(math.pi)}")
  print(f"tan(pi/4) -> {math.tan(math.pi/4)}")
  print()

  print(f"Funciones inversas: ")
  print(f"asin(1)  -> {math.asin(1)}")
  print(f"acos(-1) -> {math.acos(-1)}")
  print(f"atan(1)  -> {math.atan(1)}")
  print()

main()
```

2.4. Exponencial, Logaritmos, Potenciación y Raíz Cuadrada

2.4.1. Funciones

```python
''' Función exponencial '''
math.exp(x)   # Base e
math.exp2(x)  # Base 2

''' Logaritmos '''
math.log(x)    # Logaritmo natural (base e)
math.log10(x)  # Logaritmo base 10
math.log2(x)   # Logaritmo base 2

''' Potenciación '''
math.pow(b, e) # b elevado a e

''' Raíz cuadrada '''
math.sqrt(x)
```

2.4.2. Ejemplos

```python
import math

def main():
  x = float(input("Ingrese un número: "))
  print(f"e elevado a la {x} es {math.exp(x)}")

main()
```

2.5. Constantes

Para evitar cálculos innecesarios o simplemente para hacer uso de algunos valores útiles, la biblioteca matemática tiene las siguiente **constantes**.

```
math.e     # e   -> 2.7182818284590452540
math.pi    # pi  -> 3.14159265358979323846
math.tau   # tau -> 6.283185307179586
```

3. Comprobación de caracteres

Los *strings* poseen algunas funciones que ayudan a determinar algunas características que podrán ser aplicadas para algunos algoritmos que se estudiarán en la próxima unidad. Para invocarlas debemos aplicar el operador '`.`' (punto) junto al nombre de la variable del tipo `str` por ejemplo:

```
>>> mi_texto = "un texto"
>>> mi_texto.upper()
'UN TEXTO'
>>>
```

3.1. Funciones

```
''' Funciones booleanas '''
islower()     # True si todos los caracteres son minúscula
isupper()     # True si todos los caracteres son mayúscula
isalnum()     # True si todos los caracteres son alfanuméricos
isdecimal()   # True si todos lo caracteres son numéricos (0 a 9)
isalpha()     # True si todos los caracteres son letras
isprintable() # True si los caracteres son imprimibles y espacio
isspace()     # True si ' ','\t','\v','\r','\n','\f'
''' procesamiento de datos '''
upper() # Devuelve el string en mayúsculas
lower() # Devuelve el string en minúsculas
```

3.2. Ejemplos

```
1 def main():
2   c = 'c'.upper()    # se guarda el valor 'C'
3   print(f"Mayúscula de 'c' es '{c}'")
4   g = 'G'.lower()    # se guarda el valor 'g'
5   print(f"Minúscula de 'G' es '{g}'")
6   if 'h'.isalpha():
7     print("'h' es letra...")
```

```
8
9     if '9'.isdecimal():
10      print("'9' es número...")
11
12  main()
```

4. Manejo de entrada/salida segura

Aquí se expondrá una técnica para evitar errores al momento de validar datos ingresados por el usuario, es decir lo que ocurre cuando ingresan un carácter en vez de un número, ya sea entero o decimal.

4.1. Excepciones

En `Python` hay una filosofía: *"es mejor pedir perdón que pedir permiso"*. Eso quiere decir que ante un error todo termina abruptamente. El mecanismo que utiliza es uno muy habitual en muchos lenguajes de programación conocido como **excepciones**. Esto está asociado a un mecanismo de sintaxis habitualmente conocido como *try-catch* donde el programador resguarda en el ámbito de un bloque `try` la porción de código que posiblemente de error. Luego, *"ataja"* el error o excepción ocacionado de surgir. En este sentido es *similar* a una sentencia de selección simple.

Observemos qué nos dice el `REPL` cuando ingresamos una letra al momento de pedir ingresar un número:

```
>>> n = int(input("Ingrese un número enetero: "))
Ingrese un número enetero: f
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'f'
>>>
```

Hay que prestar especial atención al tipo de error, en este caso `ValueError`. Ahora se implementarán funciones para ingresar números que no puedan fallar utilizando el mecanismo de excepciones:

```
1  '''
2    Muestra un mensaje, si no se ingresa un int se muestra
3    un mensaje de error y se pide ingresar otro valor, hasta
4    que sea correcto.
5  '''
6  def pedir_entero(msj):
7    while True: # bucle infinito
8      try:
9        x = int(input(msj)) # esto puede "arrojar" un ValueError
10       return x            # si no hubo error se devuelve el valor
11     except ValueError:    # si no se ingresó un int:
12       print("Error de ingreso, intente nuevamente...")
13     # se repite el bucle
```

```
14
15  '''
16    Muestra un mensaje, si no se ingresa un float se muestra
17    un mensaje de error y se pide ingresar otro valor, hasta
18    que sea correcto.
19  '''
20  def pedir_numero(msj):
21    while True: # bucle infinito
22      try:
23        x = float(input(msj)) # esto puede "arrojar" un ValueError
24        return x              # # si no hubo error se devuelve el
     valor
25      except ValueError:      # si no se ingresó un float:
26        print("Error de ingreso, intente nuevamente...")
27      # se repite el bucle
28
29  def main():
30    n = pedir_entero("Ingrese un número entero: ")
31    print(n)
32    n = pedir_numero("Ingrese un número cualquiera: ")
33    print(n)
34
35  main()
```

Como podemos observar, en las *líneas 10 y 25*, se hace una lectura como las que ya se utilizaban, pero en el caso de que haya un error de valor (`ValueError`) producido porque en vez de un número válido se ingresa otra cosa, entonces se saltará directamente a la *línea 13 ó 28* respectivamente. Como el `try-except` se encuentra dentro de un bucle infinito (`while True`), si ocurre el error se volverá a pedir al usuario un ingreso. En caso de que se pueda guardar el valor exitosamente, entonces el bucle termina porque se encuentra con la sentencia `return`, devolviendo el valor leído exitosamente.

5. Utilitarias

Hay una cantidad basta de funciones en los módulos principales de `Python`. Aquí se expondrán solo algunas que podrían ser de interés para ciertos algoritmos.

5.1. Tiempo

Dentro de los módulos estándar se encuentran funciones de manejo de tiempo a los cuales se puede acceder al importar `time`. Será de interés para el curso utilizar las funciones `time()` y `sleep()`. La función `time()` retornará el valor calendario actual del sistema sobre el cual se está ejecutando en un formato numérico entero y representan los segundos transcurridos desde *"The Epoch"* que es la hora 00:00UTC del 1 de enero de 1970[1].

[1]Para más información puede consultar en https://es.wikipedia.org/wiki/Tiempo_Unix

Programa 3: Leer el tiempo calendario del sistema.

```
import time

def main():
  s = time.time()

  print(f"Han transcurridos {s} segundos "
        "desde las 0hs UTC del 1 de enero de 1970.")

main()
```

Este dato es útil para contabilizar duraciones, implementar una aplicación que necesite de marcas de tiempo, agendas, calendarios, etc. Por ejemplo, podremos saber cuánto tiempo estuvo ejecutándose un programa.

```
import time

def main():
  init = time.time()

  ''' ... resto del programa ... '''

  print( "Tiempo de ejecución "
        f"{time.time()-init}"      # calculo tiempo transcurrido
         " segundos.")

main()
```

Por otro lado, la función `sleep()` nos da la posibilidad de generar *demoras* en nuestro código. Esto podría ser útil en caso de mostrar información espaciadamente al usuario, sin la necesidad de su interacción.

Programa 4: Escribir segundos transcurridos

```
import time

def main():
  for i in range(10):
    time.sleep(1); # esperar 1 segundo
    print(f"{i+1}s transcurrido(s)...")

  print"Fin del programa")

main()
```

5.2. Números aleatorios

En algunas ocasiones será de interés generar números *al azar*, ya sea porque son parte de un algoritmo o para generar diferentes comportamientos cada vez que se ejecute nuestro programa.

En **Python** se pueden obtener muy fácilmente números *pseudo-aleatorios*. Se los denomina así ya que necesita de una *semilla* o valor inicial para poder ser generados utilizando la función `seed()`. Si la semilla es siempre la misma los números generados seguirán el mismo patrón, es decir, que serán los mismos en cada ejecución. Si se omite llamar a esta función o se llama sin un argumento, utilizará el tiempo actual del reloj o la unidad de generación del sistema operativo. En la mayoría de los casos, simplemente podremos optar por no llamar a esta función y utilizar directamente la función `random()`. Para poder acceder a estas debemos importar `random`.

Programa 5: Generar un número aletario

```python
import random

def main():
  # La semilla para generar los número aleatorios es tomada
  # del tiempo actual de la PC que ejecuta el programa.
  random.seed() # podemos omitir esto
  # Se genera un número aleatorio
  n = random.random();
  print(f"Número generado aleatoreamente: {n}")

main()
```

Como se puede comprobar los valores generados son reales entre $[0{,}0; 1{,}0)$. Si se desea otro rango o números enteros podemos utilizar otra función llamada `randint(a,b)` donde obtendremos un número n que cumple con $\forall a, b, n \in \mathbb{Z} / a \leq n \leq b$.

Programa 6: Números pseudo-aleatorios enteros

```python
import random

def main():
  # Se genera un número aleatorio entero entre [5;-5]
  n = random.randint(-5, 5);
  print(f"Número generado aleatoreamente: {n}")

main()
```

6. Ejercitación

Intente resolver los siguientes problemas planteando los algoritmos en diagramas y codifíquelos en **Python validando siempre las entradas por parte del usuario**, volviendo a pedir los datos ingresados en caso de error.

1. Pida ingresar dos números reales a y b, ambos positivos o cero. Calcule el resultado de $\sqrt{a^b}$ y muéstrelo en pantalla redondeado a 2 decimales.

2. Pida ingresar los valores de 2 puntos en el plano (x_1, y_1) y (x_2, y_2). Calcule y muestre la distancia entre dichos puntos redondeados a 3 decimales aplicando el teorema de pitágoras $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

3. Pida ingresar un ángulo en radianes y el lado adyacente de un triangulo rectángulo. Calcule y muestre por pantalla el valor de la la hipotenusa y el opuesto de dicho triangulo. Recuerde que: $\sin(\theta) = \frac{OP}{HIP}$; $\cos(\theta) = \frac{AD}{HIP}$; $\tan(\theta) = \frac{OP}{AD}$

4. Repita el punto anterior agregando una opción previa al usuario con la cual decida si quiere ingresar el ángulo en radianes o grados. Y en caso de que elija grados haga la conversión pertinente para el cálculo.

5. Pida ingresar 3 variables reales a, b y c pertenecientes a una función cuadrática ($a.x^2 + b.x + c$) e indique si posee raíces reales (2 simples o una doble) o no posee raíces reales. Calcule y muestre dichas raíces.

6. Pida al usuario ingresar el radio r y el centro en el plano $c(h, k)$ de una circunferencia. Calcule y muestre su perímetro y área. Luego pida ingresar un punto cualquiera $p(x, y)$ e indique si se encuentra dentro, fuera o en el perímetro del círculo.

7. Implemente un programa que pida una cantidad de dados y de lados de dichos dados. Luego, *tire los dados*, muestre el valor de cada uno y el valor acumulado.

8. Implemente el programa del punto anterior agregando *cantidad de tiradas* y ejecútelas cada 5 segundos hasta finalizar. Los valores acumulados a mostrar deben ser de cada tirada, **no es relevante** el valor acumulado total de todas las tiradas.