Status	Finished
Started	Friday, 22 November 2024, 11:05 AM
Completed	Friday, 22 November 2024, 11:12 AM
Duration	7 mins 31 secs
Marks	8.33/9.00
Grade	**9.26** out of 10.00 (**92.59**%)

Information

Tipos de datos

Los programas manejan datos usando variables. Wikipedia define un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable cuantitativa o cualitativa".

Existen datos de diferentes tipos: texto, como el nombre de una persona, números, como la edad de una persona, fechas, como la fecha de nacimiento de una persona, etc.

Los tipos de datos estándar de Python incluyen números (enteros, reales, complejos y booleanos), strings (texto), tuplas, listas y diccionarios. Las strings, las tuplas y las listas se agrupan en la categoría de secuencias.

Por ahora, nos centraremos en los números y las strings.

Information

Números

Los números en Python pueden ser: enteros (int), reales (float) y complejos (complex). La siguiente tabla muestra ejemplos de literales numéricos para expresar valores de cada uno de estos tipos:

`int`	`float`	`complex`
`10`	`0.0`	`3.14j`
`100`	`15.20`	`45.j`
`-786`	`-21.9`	`9.322e-36j`
`0o70`	`32.3e+18`	`.876j`
`-0o490`	`-90.`	`-.6545+0J`
`-0x260`	`-32.54e100`	`3e+26J`
`0x69`	`7E-12`	`4.53e-7j`

Para los literales de int, un `0o` o `0O` antes de las cifras indica que el resto de la secuencia representa un número en base 8 (octal), en lugar de la habitual base 10 (decimal), y el prefijo `0x` o `0X` indica que está en base 16 (hexadecimal); en cualquier caso, el valor numérico se representará internamente por el número en base 2 que corresponda (p.e. el mismo para `10`, `0o12` y `0xa`, ya que todos ellos representan el valor entero diez). El punto o/y la `e`/`E` (que se lee "por diez elevado a") hace que un literal numérico represente un valor float. La unidad imaginaria `j`/`J`, que tiene que venir precedida de un número (incluso `0`), hace que sea de complex.

Todos los tipos numéricos admiten las siguientes operaciones (aunque por simplicidad se han usado enteros en los ejemplos):

Operación	**Resultado**
`x + y`	Suma de x más y: 5 + 2 da 7
`x - y`	Resta de x menos y: 5 - 2 da 3
`x * y`	Producto de x por y: 5 * 2 da 10
`x / y`	Cociente de dividir x entre y: 5 / 2 da 2.5
`x // y`	Cociente entero de x entre y: 5 // 2 da 2 *El número de veces que el divisor cabe entero en el dividendo: 3.14 // 0.7 da 4.0* *(Si el dividendo o el divisor son float, el resultado es float, si ambos son int, el resultado es int)*
`x % y`	Resto de x // y: 5 % 2 da 1; 3.14 % 0.7 da 0.34 *Se mantiene el signo del divisor : 5 % -2 da -1*
`-x`	x negada: 5 si x vale -5
`+x`	x sin cambio: -5 si x vale -5
`abs(x)`	Valor absoluto de x: abs(-5) da 5
`divmod(x, y)`	La pareja (x // y, x % y): divmod(5, 2) da (2, 1)
`pow(x, y)`	x elevado a y: pow(5, 2) da 25
`x ** y`	x elevado a y: 5 ** 2 da 25

Question 1

Complete

Mark 0.33 out of 1.00

¿Cuáles de los siguientes identificadores representan un tipo básico de número en Python?

Select one or more:

- [x] int
- [ ] float
- [ ] natural
- [ ] complex

Information

Valores booleanos

El tipo bool solo dispone de dos valores, que representamos por los literales *True* y *False*. En Python se le considera un subtipo de int, y los siguientes valores numéricos se toman como *False* al emplearse donde se espera normalmente un bool (en caso contrario, se tomaría el valor *True*):

- El valor cero de cualquier tipo numérico: 0 (int), 0.0 (float), 0j (complex)
- Una secuencia vacía, p.e. la string "", compuesta de cero caracteres (aunque no por ello deja de ser una string)
- El valor que representamos por el literal *None*, un valor especial que, aunque significa "ningún valor", ¡es un valor!, de hecho es el único valor del tipo *NoneType*

En general, a cualquier valor x del tipo que sea podemos aplicarle bool(x), que nos dará un valor bool consistente con lo dicho. Naturalmente, bool(False) da False y bool(True) da True.

Los operadores **or**, **and** y **not** esperan normalmente operandos de tipo bool:

Operación	Resultado
`x or y`	Si ambos son bool, el resultado es *False* solo si **los dos** son *False*. Las operandos se evalúan de izquierda a derecha, de manera que si bool(x) es *True*, el resultado es el valor de x, no evaluándose y; en caso contrario el resultado será el valor de y
`x and y`	Si ambos son bool, el resultado es *True* solo si **los dos** son *True*. Las operandos se evalúan de izquierda a derecha, de manera que si bool(x) es *False*, el resultado es el valor de x, no evaluándose y; en caso contrario el resultado será el valor de y
`not x`	Negación: si bool(x) da *False*, el resultado es *True*, si no, es *False*

Por el funcionamiento de or y and aquí descrito se dice que son operadores **cortocircuito**, porque a diferencia de lo usual el valor de un operando puede "cortocircuitar" (evitar la evaluación) de otro(s) operando(s); lo cual con cierta frecuencia es útil, en particular si la evaluación cortocircuitada en ese caso produciría un error.

Question 2

Complete

Mark 1.00 out of 1.00

¿Cuáles de las siguientes operaciones dan como resultado *True*?

Select one or more:

- [ ] False and False
- [x] True and True
- [x] True or True
- [ ] False or False
- [ ] True and False
- [x] False or True
- [ ] not True
- [x] not False
- [ ] False and True
- [x] True or False

Information

Strings

Las strings representan secuencias contiguas de caracteres. Los literales de valores de tipo string (str), se encierran entre comillas dobles o simples:

```
var1 = "hola"
var2 = 'hola'
```

```
str_var = "Hello world"
print(str_var)          # Muestra "Hello world"
```

La longitud de una string (número de caracteres) se obtiene programáticamente usando la función len():

```
str_var = "Hello world"
print(len(str_var))     # Muestra el valor entero 11
```

Se puede acceder a un carácter concreto de una string escribiendo la string, o el nombre de la variable que la referencia, seguida del índice de la posición que ocupa el carácter deseado entre corchetes. El índice de la primera posición es 0.

```
str_var = "Hello world"
print(str_var[0])       # Muestra el primer carácter de str_var: H
print(str_var[4])       # Muestra el quinto carácter de str_var: o
```

También se pueden usar índices relativos a la longitud de la string:

```
str_var = "Hello world"
print(str_var[-1])      # Muestra el último carácter de str_var: d
print(str_var[-5])      # Muestra el quinto carácter, empezando por el final, de str_var: w
```

Se puede obtener una substring (subsecuencia) de una string usando el operador de segmento o *slice* `[:]`. El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final; si se omite el primero, toma desde el principio hasta la posición del índice anterior al indicado:

```
str_var = "Hello world"
print(str_var[2:5])     # Muestra los caracteres de 3º al 5º: "llo"
print(str_var[2:])      # Muestra los caracteres a partir del 3º: "llo world"
print(str_var[:5])      # Muestra los caracteres hasta el 5º: "Hello"
```

El diccionario de la RAE define **concatenar** como "**unir dos o más cosas**"; en el caso de las strings, concatenar dos strings consiste en formar una nueva string juntando, en orden, dos preexistentes.

En Python existen dos operadores de concatenación: el signo más (+) es el operador de concatenación de strings y el asterisco (*) es el operador de repetición (equivalente a realizar una concatenación de la misma string repetidas veces).

```
str_var = "Hello world"
print('a' + 'a')        # Muestra "aa"
print(str_var + "TEST") # Muestra el valor de str_var concatenado con "TEST": "Hello worldTEST"
print('ab' * 5)         # Muestra "abababababab"
print(str_var * 2)      # Muestra "Hello worldHello world"
```

Podríamos decir que la repetición de strings es a la concatenación de strings lo que la multiplicación de números enteros es a la suma de números enteros.

Nótese que la concatenación junta directamente las strings, sin añadir ningún carácter en medio.

Question 3

Complete

Mark 1.00 out of 1.00

¿Qué resultado muestra la siguiente operación?

```
abecedario = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
print(abecedario[5:10])
```

Answer: FGHIJ

Question 4

Complete

Mark 1.00 out of 1.00

¿Qué resultado muestra la siguiente operación?

```
print("hola" + "mundo")
```

Select one:

- "holamundo"
- "hola, mundo"
- "hola mundo"

Information

Conocer el tipo de un dato

Podemos conocer el tipo de un dato usando la operación *type()*.

```
a = 200
b = "200"
t1 = type(a)
t2 = type(b)
print(t1)
print(t2)
```

```
<type 'int'>
<type 'str'>
```

Nótese que el valor devuelto por *type()* es un valor de tipo *type*:

```
print(type(t1))
```

```
<type 'type'>
```

El valor, aunque se muestra con el formato de los ejemplos, es uno de los tipos de datos (*int*, *float*, *str*, ...):

```
print(t1 == int)
```

```
True
```

Si queremos obtener una string con solo el nombre del tipo, podemos consultar el atributo *__name__* (un atributo es una propiedad asociada a un dato):

```
typeName = t1.__name__
print(typeName)
print(type(typeName))
```

```
int
<type 'str'>
```

Question 5

Complete

Mark 1.00 out of 1.00

¿Qué valor se muestra?

```
a = 123.0
t1 = type(a)
print(t1)
```

- 'float'
- <type 'float'>
- <type float>
- <type 'int'>

Information

Type casting

En programación se conoce como "type casting" a obtener, a partir de un valor de un tipo T1, un valor "similar" de otro tipo T2; lo que no en todos los casos será posible. Si T1 y T2 son el mismo tipo, es trivial, obtenemos un valor igual de ese tipo.

int() construye un número entero a partir de: un número real (*truncándolo*, esto es, eliminando la parte fraccionaria), o una string (siempre que la string represente un número entero)

```
a = int(10)    # a toma el valor entero 10
b = int(3.7)   # b toma el valor entero 3
c = int("40")  # c toma el valor entero 40
s = "25"
d = int(s)     # d toma el valor entero 25
```

float() construye un número real a partir de: un número entero, o una string (siempre que la string represente un número real o un número entero)

```
a = float(10)    # a toma el valor real 10.0
b = float(3.5)   # b toma el valor real 3.5
c = float("40")  # c toma el valor real 40.0
s = "25.0"
d = float(s)     # d toma el valor real 25.0
```

complex() construye un número complejo a partir de: un número entero, un número real, o una string (siempre que la string represente un número real, un número entero o un número complejo)

```
a = complex(10)    # a toma el valor complejo 10+0j
b = complex(3.5)   # b toma el valor complejo 3.5+0j
c = complex("40")  # c toma el valor complejo 40+0j
s = "25"
d = complex(s)     # d toma el valor complejo 25+0j
e = complex(1+2j)  # e toma el valor complejo 1+2j
```

str() construye una string a partir de casi cualquier otro tipo de datos.

```
a = str(10)    # a toma el valor string "10"
b = str(3.5)   # b toma el valor string "3.5"
c = str("40")  # c toma el valor string "40"
s = "25"
d = str(s)     # d toma el valor string "25"
e = str(1+2j)  # e toma el valor string "1+2j"
```

Information

Anotación de tipos

Python es un lenguaje de programación con *tipado dinámico*, lo que significa que el tipo de una variable (el tipo del dato asociado a la variable) se establece en el momento de asignarle un valor, tipo que cambiará si al asignarle un nuevo valor este fuera de un tipo diferente; a diferencia de los lenguajes con *tipado estático*, en los que se requiere declarar de qué tipo es cada variable, la cual solo podrá contener valores de dicho tipo (lo que podrá requerir casting según el caso; de lo contrario resultará en una situación de error intentar asignarle un valor de tipo distinto). Veamos las siguientes asignaciones en Python:

```
a = 0   # el tipo de a es int
b = 0.0 # el tipo de b es float
c = "0" # el tipo de c es str
a = 0.0 # el tipo de a cambia a float
```

En Python, no obstante, es posible "**anotar**" el tipo de una variable, lo que en principio solo tiene un efecto informativo, equivalente a un comentario, acerca del tipo de valor que se espera que contenga:

```
a: int = 0     # el tipo de a es (y debiera seguir siendo) int
b: float = 0.0 # el tipo de b es (y debiera seguir siendo) float
c: str = "0"   # el tipo de c es (y debiera seguir siendo) str
```

A diferencia de los lenguajes con tipado estático, en Python a una variable que haya sido anotada con un tipo se le puede asignar más tarde un valor de un tipo diferente; aunque hay herramientas que pueden analizar el código y señalar situaciones que supongan riesgo de que durante la ejecución algún contenido no se ajuste a la intención manifestada en la anotación.

Question 6

Complete

Mark 2.00 out of 2.00

Anote los tipos de las siguientes variables de acuerdo con el valor que se les asigna:

```
var1: float = 37.0
var2: int   = 37
```

Information

Operadores de comparación

Estos operadores son aplicables a muchos tipos de datos; comparan los valores de sus dos operandos y devuelven un valor booleano en función de la relación entre ellos, por lo que técnicamente se les llama **operadores relacionales**.

Supongamos, en el siguiente ejemplo, que la variable *a* vale 10 y la variable *b* 20:

operador	descripción	ejemplo
==	compara si son iguales	a == b da False
!=	compara si son distintos	a != b da True
>	compara si el primero es mayor que el segundo	a > b da False
<	compara si el primero es menor que el segundo	a < b da True
>=	compara si el primero es mayor o igual que el segundo	a >= b da False
<=	compara si el primero es menor o igual que el segundo	a <= b da True

Question 7

Complete

Mark 1.00 out of 1.00

Marque los que NO sean operadores relacionales en Python.

Select one or more:

- [ ] >=
- [x] =<
- [ ] <=
- [x] =
- [ ] ==

Information

Precedencia de operadores

Una expresión es una combinación de operandos y operadores. En el siguiente ejemplo, + es un operador obivamente de adición, y los números 3 y 8 son sus operandos (también podrían ser variables o cualquier otro elemento del lenguaje que produjera un valor, en este caso aritmético):

```
3 + 8
```

La expresión anterior es una expresión bastante simple, pues solo tiene un operador (podría no tener ninguno y constar de un único "operando"). En una expresión puede haber más de un operador; cuando esto ocurre, las reglas de **precedencia** determinan el orden de aplicación de los distintos operadores. Por ejemplo, la multiplicación (representada por el *) tiene una precedencia mayor que la adición, por tanto, la expresión

```
3 + 8 * 2
```

dará como resultado 19, no 22, que sería lo que daría si los operadores se aplicaran en el orden en que están escritos, o sea, de izquierda a derecha. Más precisamente, dicho valor 19 se obtiene aplicando en primer lugar el operador *, ya que tiene mayor precedencia, calculándose 8 * 2, lo que nos da el valor 16, y a continuación aplicando el operador +, empleando el resultado anterior como su operando por la derecha (los operandos, como vemos, pueden ser simples o resultantes de cálculos previos), calculándose 3 + 16, obteniéndose finalmente el valor resultado 19. Este mismo esquema se repetiría para los distintos operadores de acuerdo con las reglas de precedencia en expresiones con más operandos.

No obstante, el orden de las operaciones predefinido se puede modificar usando paréntesis (podemos usar cuantos sean convenientes), forzando en el siguiente ejemplo a calcular primero 3 + 8 a pesar de la menor precedencia de + frente a *:

```
(3 + 8) * 2
```

El siguiente listado muestra parte del orden de precedencia predefinido para los operadores de Python, donde x indica el operando para los operadores *unarios*, y los demás son *binarios*, esto es, requieren de dos operandos (los operadores binarios típicamente son *infijos*, o sea, van entre sus operandos, p.e. 0 not in v).

Cuanto **más arriba mayor precedencia**:

```
 1. **
 2. +x, -x, ~x
 3. *, /, //, %
 4. +, -
 5. <<, >>
 6. &
 7. ^
 8. |
 9. ==, !=, >, >=, <, <=, is, is not, in, not in
10. not x
11. and
12. or
```

Obsérvese que algunos operadores comparten símbolo, p.e. el -, que puede ser unario (alta precedencia) o binario (de algo menor precedencia). Por tanto, suponiendo que la variable n tenga el valor 2, la expresión -n-3 se calcula aplicando primero el - de la izquierda (que es unario pues no tiene operando a su izquierda, y por tanto tiene mayor precedencia), obteniendo el valor -2, y a continuación aplicando el - binario restándole 3, resultando -5. Naturalmente, podríamos escribir -(n-3), con lo que forzamos primero el cálculo entre paréntesis, obteniendo el valor -1, a lo que le aplicamos el - unario resultando 1. Las reglas de precedencia aplican siempre, por tanto incluso podríamos escribir 3*-n, y como el - es unario (a su izquierda no tiene operando, sino el operador *), tiene mayor precedencia que *, por lo que se obtendría primero el valor -2, realizándose a continuación la multiplicación, resultando -6.

Information

Asociatividad de operadores

La siguiente lista muestra la **precedencia** relativa de parte de los operadores en Python (x indica el operando para los operadores unarios):

```
 1. **
 2. +x, -x, ~x
 3. *, /, //, %
 4. +, -
 5. <<, >>
 6. &
 7. ^
 8. |
 9. ==, !=, >, >=, <, <=, is, is not, in, not in
10. not x
11. and
12. or
```

Como se puede observar, hay algunos operadores que tienen igual precedencia y que naturalmente podrían aparecer en una misma expresión. También puede ocurrir que el mismo operador (p.e. el - binario) aparezca varias veces en la expresión, lógicamente en todas las ocurrencias con su (igual) precedencia. En estos casos de igual precedencia esta no nos sirve para determinar completamente el orden de cálculo, por lo que se acude a las reglas de **asociatividad**. La gran mayoría de los operadores binarios son **asociativos por la izquierda**, o sea, dentro de un mismo nivel de precedencia se aplican empezando por el que esté más a la izquierda, y siguiendo hacia la derecha. Ejemplo:

> `9 - 2 * 2 + 3` es igual (* mayor precedencia) a `9 - 4 + 3`, que es igual (- binario asociativo por la izquierda) a `5 + 3`, que es igual a 8.

Si la asociatividad de los operadores binarios + y - fuese de derecha a izquierda, el resultado de la expresión anterior sería:

> `9 - 2 * 2 + 3` es igual a `9 - 4 + 3`, que sería igual a `9 - 7`, que sería igual a 2: ¡un disparate!

Por conveniencia, el operador de exponenciación en particular es asociativo por la **derecha**:

> `2 ** 3 ** 2` es igual a `2 ** 9` que es igual a 512 (interpretamos que el 2 de la izquierda está elevado a 3-elevado-a-2)

Si ** tuviera asociatividad de izquierda a derecha el resultado sería:

> `2 ** 3 ** 2` sería igual a `8 ** 2` que sería igual a 64: no habríamos considerado que 3 está elevado a 2)

Tiene poco sentido que los operadores de comparación sean asociativos, y en Python se interpretan de forma especial, p.e.:

> `a < b < c == d`
>
> **se interpreta** como sigue (si bien b y c, aunque se muestren repetidos, se evaluarían una sola vez):
>
> `a < b and b < c and c == d`

Obsérvese que los and tienen menor precedencia y sí son asociativos (por supuesto por la izquierda, además de cortocircuito).

Question 8

Complete

Mark 1.00 out of 1.00

Empareje cada expresión con su resultado:

2 * 5 + 3 - 8 + 2 ** 2 ** 3	261
2 * (5 + 3) - 8 + 2 ** (2 ** 3)	264
2 * (5 + 3 - 8) + 2 ** 2 ** 3	256