In [34]:

```
#Q NO 01
from math import pi, e, cos, sin, tan, sqrt, asinh, acosh, tanh, log
print("cos zero is", cos(0))
print("sin 30 is", sin(30))
print("tan 45 is", tan(45))
print("asinh 60 is", asinh(60))
print("acosh 90 is", acosh(90))
print("tanh 180 is", tanh(180))
print("log of 10 is", log(10))
print("cosine pi is", cos(pi))
print("log base e is", log(e))
print("square root of 169 is", sqrt(169))

print("time taken 10 min")
```

```
cos zero is 1.0
sin 30 is -0.9880316240928618
tan 45 is 1.6197751905438615
asinh 60 is 4.78756117999381
acosh 90 is 5.192925985263684
tanh 180 is 1.0
log of 10 is 2.302585092994046
cosine pi is -1.0
log base e is 1.0
square root of 169 is 13.0
time taken 10 min
```

In [32]:

```
# Q NO 03
my_name = ("saad")
print(len(my_name))
print("time taken 3mins")
```

```
4
time taken 3mins
```

In [16]:

```
# Q.NO 04


name = ("saad")
print(type(name))
birth_date = ("24 July 2001")
print(type(birth_date))
cell_no = ("03327974269")
print(type(cell_no))
email = ("saudnoor04@gmail.com")
print(type(email))
favourite_dish = ("qourma")
print(type(favourite_dish))
favourite_colour = ("black")
print(type(favourite_colour))
favourite_movie = ("joker")
print(type(favourite_movie))
favourite_flower = ("jasmine")
print(type(favourite_flower))
print("time taken: 6min")
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

```
time taken: 6min
```

In [24]:

```python
#Q NO. 05

# A
import math
angle_degrees = 30
v = 15
g = 9.8
a = math.sin(30)
x = v**2*a**2
y = 2*g
h = x/y
print("the maximum height above the ground is :" + str(h) + " meters ")

# B
v = 15
g = 9.8
a = sin(math.degrees(30))
b = 2*v
c = b*a
t = -c/g
print("the total time in the air :" +str(t) + " sec ")

# C
v = 15
t = 3.024
g = 9.8
s = -(v*t*math.sin(30) - 1/2*g*t**2)
print("the total distance travelled in :" + str(s) + " meters ")

# D
v = 15
g = -9.8
t = 3.024
v = v*sin(30)-g*t
print("the velocity vector at max.height is :" + str(v) + " meters/second ")

#E
from math import sqrt
v = 15
g = 9.8
h = 11.2
t = sqrt(2*h/g)
final_v = v+g/t
print("the velocity when ball hits the ground is :" + str(final_v) + " meters/second ")

print("timetaken 18min")
```

```
the maximum height above the ground is :11.20645205595434 meters
the total time in the air :1.25440346502288 sec
the total distance travelled in :89.62553686885221 meters
the velocity vector at max.height is :14.814725638607074 meters/second
the velocity when ball hits the ground is :21.48209071210825 meters/second
timetaken 18min
```

In [27]:

```python
# Q NO.07

r1 = 5
r2 = 8
r3 = 7
v = 10
total_r = (r1+r2+r3)
print(total_r)
I = v/total_r
v1 = I*r1
v2 = I*r2
v3 = I*r3
print("total current across series circuit: ",I)
print("voltage across r1: ",v1)
print("voltage across r2: ",v2)
print("voltage across r2: ",v2)
print("voltage across r3: ",v3)

#in parallel
total_resistace = 1/(1/r1 +1/r2 +1/r3)
I = v/total_r
I1 = v/r1
I2 = v/r2
I3 = v/r3
print("total current across parallel circuit: ",I)
print("current across r1: ",I1)
print("current across r2: ",I2)
print("current across r3: ",I3)

print("time taken : 13 mins")
```

```
20
total current across series circuit:  0.5
voltage across r1:  2.5
voltage across r2:  4.0
voltage across r3:  3.5
total current across parallel circuit:  0.5
current across r1:  2.0
current across r2:  1.25
current across r3:  1.4285714285714286
time taken : 13 mins
```

In [28]:

```
# Q NO.9

eggs = 120
milk = 95
butter = 0
mayonise = 124
cookies = 456
drinks = 250
icecream = 350
snacks = 75
rice = 125
salt = 20

a = eval(input("eggs are 120 per dozen , enter the number of eggs you want :"))
b = eval(input("milk is 95 per liter , enter the liter you want :"))
c = eval(input("butter is 100 per gram , enter the quantity you want :"))
d = eval(input("mayonise is 124 per gram , enter the quantity you want :"))
e = eval(input("cookies are 456 per box , enter the amount quantity you want :"))
f = eval(input("drinks are 250 per carton , enter the quantity you want :"))
g = eval(input("icecream are 350 per box , enter the quantity you want :"))
h = eval(input("snacks are 75 per kg , enter the quantity you want :"))
i = eval(input("rice are 75 per kg , enter the quantity you want :"))
j = eval(input("salt is 20 , enter the quntity you want :"))
total = (a*120 + b*95 + c*100 + d*124 + e*456 + f*250 + g*350 + h*75 + i*125 + j*20 )

print ("\n\ntotal amount to be paid : ", total, "RS")

print("time taken 13min")
```

```
eggs are 120 per dozen , enter the number of eggs you want :30
milk is 95 per liter , enter the liter you want :3
butter is 100 per gram , enter the quantity you want :2
mayonise is 124 per gram , enter the quantity you want :5
cookies are 456 per box , enter the amount quantity you want :9
drinks are 250 per carton , enter the quantity you want :3
icecream are 350 per box , enter the quantity you want :5
snacks are 75 per kg , enter the quantity you want :10
rice are 75 per kg , enter the quantity you want :4
salt is 20 , enter the quntity you want :2


total amount to be paid :  12599 RS
time taken 13min
```

In [36]:

```python
# Q NO.02

(help('math.pow'))
print("time taken 6min")
```

```
Help on built-in function pow in math:

math.pow = pow(x, y, /)
    Return x**y (x to the power of y).

time taken 6min
```

In [38]:

```python
# Q NO.02

(help (int))
print("time taken 6 min")
```

```
Help on class int in module builtins:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |  
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |  
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |  
 |  Methods defined here:
 |  
 |  __abs__(self, /)
 |      abs(self)
 |  
 |  __add__(self, value, /)
 |      Return self+value.
 |  
 |  __and__(self, value, /)
 |      Return self&value.
 |  
 |  __bool__(self, /)
 |      self != 0
 |  
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |  
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |  
 |  __eq__(self, value, /)
 |      Return self==value.
 |  
 |  __float__(self, /)
 |      float(self)
 |  
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |  
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |  
 |  __format__(self, format_spec, /)
 |      Default object formatter.
 |  
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
 |      Return value+self.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rlshift__(self, value, /)
 |      Return value<<self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |      Return value|self.
 |  
 |  __round__(...)
 |      Rounding an Integral returns itself.
 |      Rounding with an ndigits argument also returns an integer.
 |  
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |  
 |  __rrshift__(self, value, /)
 |      Return value>>self.
 |  
 |  __rshift__(self, value, /)
 |      Return self>>value.
 |  
 |  __rsub__(self, value, /)
 |      Return value-self.
 |  
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |  
 |  __rxor__(self, value, /)
 |      Return value^self.
 |  
 |  __sizeof__(self, /)
 |      Returns size in memory, in bytes.
 |  
 |  __str__(self, /)
 |      Return str(self).
 |  
 |  __sub__(self, value, /)
 |      Return self-value.
 |  
 |  __truediv__(self, value, /)
 |      Return self/value.
 |  
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |  
 |  __xor__(self, value, /)
 |      Return self^value.
 |  
 |  bit_length(self, /)
 |      Number of bits necessary to represent self in binary.
 |      
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |  
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |  
 |  to_bytes(self, /, length, byteorder, *, signed=False)
 |      Return an array of bytes representing an integer.
 |      
 |      length
 |        Length of bytes object to use.  An OverflowError is raised if the
 |        integer is not representable with the given number of bytes.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
 |      signed
 |        Determines whether two's complement is used to represent the integer.
 |        If signed is False and a negative integer is given, an OverflowError
 |        is raised.
 |  
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |  
 |  from_bytes(bytes, byteorder, *, signed=False) from builtins.type
 |      Return the integer represented by the given array of bytes.
 |      
 |      bytes
 |        Holds the array of bytes to convert.  The argument must either
 |        support the buffer protocol or be an iterable object producing bytes.
 |        Bytes and bytearray are examples of built-in objects that support the
 |        buffer protocol.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
 |      signed
 |        Indicates whether two's complement is used to represent the integer.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number

time taken 6 min
```

In [39]:

```python
# Q NO.02

(help('class'))
print("time taken 5min")
```

```
Class definitions
*****************

A class definition defines a class object (see section The standard
type hierarchy):

   classdef    ::= [decorators] "class" classname [inheritance] ":" suite
   inheritance ::= "(" [argument_list] ")"
   classname   ::= identifier

A class definition is an executable statement.  The inheritance list
usually gives a list of base classes (see Metaclasses for more
advanced uses), so each item in the list should evaluate to a class
object which allows subclassing.  Classes without an inheritance list
inherit, by default, from the base class "object"; hence,

   class Foo:
       pass

is equivalent to

   class Foo(object):
       pass

The class’s suite is then executed in a new execution frame (see
Naming and binding), using a newly created local namespace and the
original global namespace. (Usually, the suite contains mostly
function definitions.)  When the class’s suite finishes execution, its
execution frame is discarded but its local namespace is saved. [3] A
class object is then created using the inheritance list for the base
classes and the saved local namespace for the attribute dictionary.
The class name is bound to this class object in the original local
namespace.
```

```
The order in which attributes are defined in the class body is
preserved in the new class’s "__dict__".  Note that this is reliable
only right after the class is created and only for classes that were
defined using the definition syntax.

Class creation can be customized heavily using metaclasses.

Classes can also be decorated: just like when decorating functions,

   @f1(arg)
   @f2
   class Foo: pass

is roughly equivalent to

   class Foo: pass
   Foo = f1(arg)(f2(Foo))

The evaluation rules for the decorator expressions are the same as for
function decorators.  The result is then bound to the class name.

**Programmer’s note:** Variables defined in the class definition are
class attributes; they are shared by instances.  Instance attributes
can be set in a method with "self.name = value".  Both class and
instance attributes are accessible through the notation “"self.name"”,
and an instance attribute hides a class attribute with the same name
when accessed in this way.  Class attributes can be used as defaults
for instance attributes, but using mutable values there can lead to
unexpected results.  Descriptors can be used to create instance
variables with different implementation details.

See also:

  **PEP 3115** - Metaclasses in Python 3000
     The proposal that changed the declaration of metaclasses to the
     current syntax, and the semantics for how classes with
     metaclasses are constructed.

  **PEP 3129** - Class Decorators
     The proposal that added class decorators.  Function and method
     decorators were introduced in **PEP 318**.

Related help topics: CLASSES, SPECIALMETHODS

time taken 5min
```