Exponentiation All Versions

Examples

Exponentiation using builtins: ** and pow()

Exponentiation can be used by using the builtin `pow` -function or the `**` operator:

```
2 ** 3    # 8
pow(2, 3) # 8
```

For most (all in Python 2.x) arithmetic operations the result's type will be that of the wider operand. This is not true for `**` ; the following cases are exceptions from this rule:

- Base: `int` , exponent: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- This is also valid for Python 3.x.
- Before Python 2.2.0, this raised a `ValueError` .
- Base: `int < 0` or `float < 0` , exponent: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Before python 3.0.0, this raised a `ValueError` .

The `operator` module contains two functions that are equivalent to the `**` -operator:

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

or one could directly call the `__pow__` method:

```
val1, val2 = 4, 2
val1.__pow__(val2)      # 16
val2.__rpow__(val1)     # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

Square root: math.sqrt() and cmath.sqrt

The `math` module contains the `math.sqrt()` -function that can compute the square root of any number (that can be converted to a `float`) and the result will always be a `float` :

```
import math

math.sqrt(9)                # 3.0
math.sqrt(11.11)            # 3.3331666624997918
math.sqrt(Decimal('6.25'))  # 2.5
```

The `math.sqrt()` function raises a `ValueError` if the result would be `complex` :

```
math.sqrt(-10)
```

> ValueError: math domain error

`math.sqrt(x)` is *faster* than `math.pow(x, 0.5)` or `x ** 0.5` but the precision of the results is the same. The `cmath` module is extremely similar to the `math` module, except for the fact it can compute complex numbers and all of its results are in the form of a + bi. It can also use `.sqrt()` :

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

What's with the `j` ? `j` is the equivalent to the square root of -1. All numbers can be put into the form a + bi, or in this case, a + bj. `a` is the real part of the number like the 2 in `2+0j` . Since it has no imaginary part, `b` is 0.

`b` represents part of the imaginary part of the number like the 2 in `2j`. Since there is no real part in this, 2j can also be written as `0 + 2j`.

Modular exponentiation: pow() with 3 arguments

Supplying `pow()` with 3 arguments `pow(a, b, c)` evaluates the modular exponentiation $a^b \bmod c$:

```
pow(3, 4, 17)   # 13

# equivalent unoptimized expression:
3 ** 4 % 17     # 13

# steps:
3 ** 4          # 81
81 % 17         # 13
```

For built-in types using modular exponentiation is only possible if:

- First argument is an `int`
- Second argument is an `int >= 0`
- Third argument is an `int != 0`

These restrictions are also present in python 3.x

For example one can use the 3-argument form of `pow` to define a modular inverse function:

```
def modular_inverse(x, p):
    """Find a such as  a·x ≡ 1 (mod p), assuming p is prime."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

Computing large integer roots

Even though Python natively supports big integers, taking the nth root of very large numbers can fail in Python.

```
x = 2 ** 100
cube = x ** 3
root = cube ** (1.0 / 3)
```

> OverflowError: long int too large to convert to float

When dealing with such large integers, you will need to use a custom function to compute the nth root of a number.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

Exponential function minus 1: math.expm1()

Exponential function minus 1: math.expm1()

The `math` module contains the `expm1()` -function that can compute the expression `math.e ** x - 1` for very small `x` with higher precision than `math.exp(x)` or `cmath.exp(x)` would allow:

```
import math

print(math.e ** 1e-3 - 1)  # 0.0010005001667083846
print(math.exp(1e-3) - 1)  # 0.0010005001667083846
print(math.expm1(1e-3))    # 0.0010005001667083417
#                            ------------------^
```

For very small x the difference gets bigger:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15))   # 1.0000000000000007e-15
#                              ^-------------------
```

The improvement is significant in scientic computing. For example the Planck's law contains an exponential function minus 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)        # 4.139080074896474e-19
planks_law_naive(100, 5000)  # 4.139080073488451e-19
#                                      ^----------

planks_law(1000, 5000)       # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#                                    ^------------
```

Exponential function: math.exp() and cmath.exp()

Both the `math` and `cmath` -module contain the Euler number: e and using it with the builtin `pow()` -function or `**` -operator works mostly like `math.exp()` :

```
import math

math.e ** 2  # 7.3890560989306495
math.exp(2)  # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

However the result is different and using the exponential function directly is more reliable than builtin exponentiation with base `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Exponentiation using the math module: math.pow()

The `math` -module contains another `math.pow()` function. The difference to the builtin `pow()` -function or `**` operator is that the result is always a `float` :

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Which excludes computations with complex inputs:

```
math.pow(2, 2+0j)
```

> TypeError: can't convert complex to float

and computations that would lead to complex results:

and computations that would lead to complex results.

```
math.pow(-2, 0.5)
```

> ValueError: math domain error

Magic methods and exponentiation: builtin, math and cmath

Supposing you have a class that stores purely integer values:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Using the builtin `pow` function or `**` operator always calls `__pow__`:

```
Integer(2) ** 2                # Integer(4)
# Prints: Using __pow__
Integer(2) ** 2.5              # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)           # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)    # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

The second argument of the `__pow__()` method can only be supplied by using the builtin- `pow()` or by directly calling the method:

```
pow(Integer(2), 3, 4)          # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)       # Integer(0)
# Prints: Using __pow__ with modulo
```

While the `math` -functions always convert it to a `float` and use the float-computation:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` -functions try to convert it to `complex` but can also fallback to `float` if there is no explicit conversion to `complex`:

```
import cmath

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__   # Deleting __complex__ method - instances cannot be cast to complex

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __float__
```

Neither `math` nor `cmath` will work if also the `__float__()` -method is missing:

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: a float is required

Roots: nth-root with fractional exponents

While the `math.sqrt` function is provided for the specific case of square roots, it's often convenient to use the exponentiation operator (`**`) with fractional exponents to perform nth-root operations, like cube roots.

The inverse of an exponentiation is exponentiation by the exponent's reciprocal. So, if you can cube a number by putting it to the exponent of 3, you can find the cube root of a number by putting it to the exponent of 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Syntax

```
value1 ** value2
```

```
pow(value1, value2[, value3])
```

```
value1.__pow__(value2[, value3])
```

```
value2.__rpow__(value1)
```

```
operator.pow(value1, value2)
```

```
operator.__pow__(value1, value2)
```

```
math.pow(value1, value2)
```

```
math.sqrt(value1)
```

```
math.exp(value1)
```

```
cmath.exp(value1)
```

```
math.expm1(value1)
```

Parameters

Remarks