

Numbers

Number types

- **Numbers are immutable and single**
- **Python supports different numerical types**
 - int (signed integers): They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
 - float (floating point real values) : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 ($2.5e2 = 2.5 \times 10^2 = 250$).
 - complex (complex numbers) : are of the form $a + bJ$, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Numbers

Python Numbers

Advertisements

[Previous Page](#)

[Next Page](#)

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example –

```
var1 = 1
var2 = 10
```

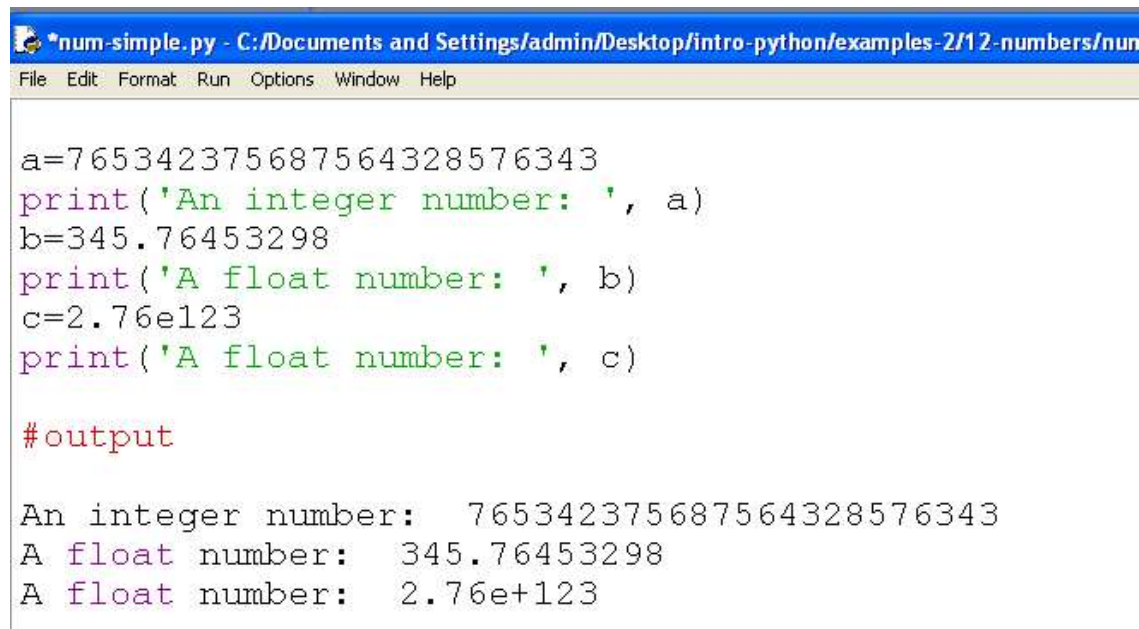
You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is –

```
del var1[,var2[,var3[...varN]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example:

```
del var
del var_a, var_b
```

Numbers



The screenshot shows a Python IDE window titled "num-simple.py - C:/Documents and Settings/admin/Desktop/intro-python/examples-2/12-numbers/nun". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code in the editor defines three variables: an integer 'a', a float 'b', and a float 'c' in scientific notation. It then prints each variable with a descriptive label. Below the code, the output of the script is displayed, showing the values of 'a', 'b', and 'c' as printed.

```
a=765342375687564328576343
print('An integer number: ', a)
b=345.76453298
print('A float number: ', b)
c=2.76e123
print('A float number: ', c)

#output

An integer number:  765342375687564328576343
A float number:    345.76453298
A float number:    2.76e+123
```

Numbers

int	float	complex
51924361	0.0	3.14j
-0x19323	15.20	45.j
0122	-21.9	9.322e-36j
0xDEFA BCECBDAECBFBAE	32.3+e18	.876j
535633629843	-90.	-.6545+0j
-052318172735	-32.54e100	3e+26j
-4721885298529	70.2-E12	4.53e-7j

Number Operators

- All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations).

Number Arithmetic Operators

Operation	Result
$x + y$	sum of x and y
$x - y$	difference of x and y
$x * y$	product of x and y
x / y	quotient of x and y
$x // y$	floored quotient of x and y
$x \% y$	remainder of x / y
$-x$	x negated
$+x$	x unchanged
$x ** y$	x to the power y

$$\begin{array}{r} 11 \overline{) 3} \\ -9 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 11 \overline{) 3} \\ -11 \\ \hline -(-9) \\ \hline -2 \end{array}$$

-3.6

$$\begin{array}{r} 11 \overline{) 3} \\ -11 \\ \hline -11 \\ \hline -2 \end{array}$$

$$\begin{array}{r} 11 \overline{) 3} \\ -11 \\ \hline -2 \\ \hline -1 \end{array}$$

$$\begin{array}{r} 3 \overline{) 11} \\ -9 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 3 \overline{) 11} \\ -9 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 3 \overline{) 11} \\ -9 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 3 \overline{) 11} \\ -9 \\ \hline 2 \end{array}$$

Number Comparison Operators

- There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, $x < y \leq z$ is equivalent to $x < y$ and $y \leq z$, except that y is evaluated only once (but in both cases z is not evaluated at all when $x < y$ is found to be false).

Number Comparison Operators

Operation	Meaning
<code><</code>	strictly less than
<code><=</code>	less than or equal
<code>></code>	strictly greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	negated object identity

Number built-in functions

- `abs(x)` : Return the absolute value of a number
- `max(x, y, z,)`
- `min(x, y, z,)`
- `round(x [, n])` : Return the floating point value *number* rounded to *ndigits* digits after the decimal point.
- `bin(x)` : Convert an integer number to a binary string.
- `hex(x)` : Convert an integer number to a lowercase hexadecimal string prefixed with “0x”,
- `pow(x, y[, z])` : Return x to the power y; if z is present, return x to the power y, modulo z
- *Id(x): Return the identity of x*

Number Type Conversion

- **Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.**
 - Type `int(x)` to convert `x` to a plain integer.
 - Type `float(x)` to convert `x` to a floating-point number.
 - Type `complex(x)` to convert `x` to a complex number with real part `x` and imaginary part zero.
 - Type `complex(x, y)` to convert `x` and `y` to a complex number with real part `x` and imaginary part `y`. `x` and `y` are numeric expressions

Number Type specific functions

- `int.bit_length()` : Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:
- `int.to_bytes(length, byteorder, *, signed=False)` : Return an array of bytes representing an integer.
- `int.from_bytes(bytes, byteorder, *, signed=False)` : Return the integer represented by the given array of bytes.
- `float.as_integer_ratio()` : Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.
- `float.is_integer()` : Return True if the float instance is finite with integral value, and False otherwise:
- `float.hex()` : Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.
- `float.fromhex(s)` : Return the float represented by a hexadecimal string.

1.5

3, 25

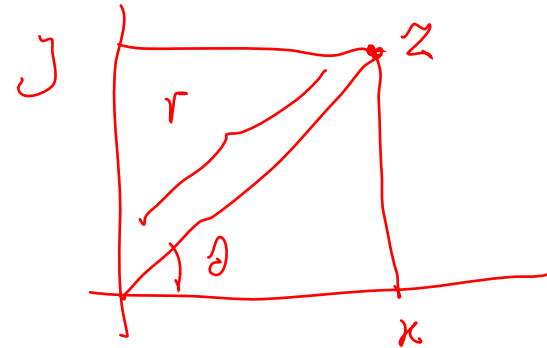
1.0

3, 0

0x1.5p1

Number Type specific functions

- `complex.real`
- `complex.image`
- `complex.conjugate`



$$z = x + jy$$
$$x = r \cos \theta$$
$$y = r \sin \theta$$

Number Type specific functions

```
a=765342375687564328576343
b=345.76453298
c=2.76e123
print('length: ', a.bit_length())
print('ratio: ', b.as_integer_ratio())
print('hex: ', c.hex())
s='0xABCDf123'
print('float: ', float.fromhex(s))

#output

length:  80
ratio:   (1520688497936193, 4398046511104)
hex:    0x1.0b3576ad6f326p+410
float:   2882400547.0
```

Complex Numbers

A complex number is a pair of real numbers a and b , most often written as $a + bi$, or $a + ib$, where i is called the imaginary unit and acts as a label for the second term. Mathematically, $i = \sqrt{-1}$. An important feature of complex numbers is definitely the ability to compute square roots of negative numbers. For example, $\sqrt{-2} = \sqrt{2}i$ (i.e., $\sqrt{2}\sqrt{-1}$). The solutions of $x^2 = -2$ are thus $x_1 = +\sqrt{2}i$ and $x_2 = -\sqrt{2}i$.

There are rules for addition, subtraction, multiplication, and division between two complex numbers. There are also rules for raising a complex number to a real power, as well as rules for computing $\sin z$, $\cos z$, $\tan z$, e^z , $\ln z$, $\sinh z$, $\cosh z$, $\tanh z$, etc. for a complex number $z = a + ib$. We assume in the following that you are familiar with the mathematics of complex numbers, at least to the degree encountered in the program examples.

Complex Numbers

let $u = a + bi$ and $v = c + di$

The following rules reflect complex arithmetics:

$$u = v \Rightarrow a = c, b = d$$

$$-u = -a - bi$$

$$u^* \equiv a - bi \quad (\text{complex conjugate})$$

$$u + v = (a + c) + (b + d)i$$

$$u - v = (a - c) + (b - d)i$$

$$uv = (ac - bd) + (bc + ad)i$$

$$u/v = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

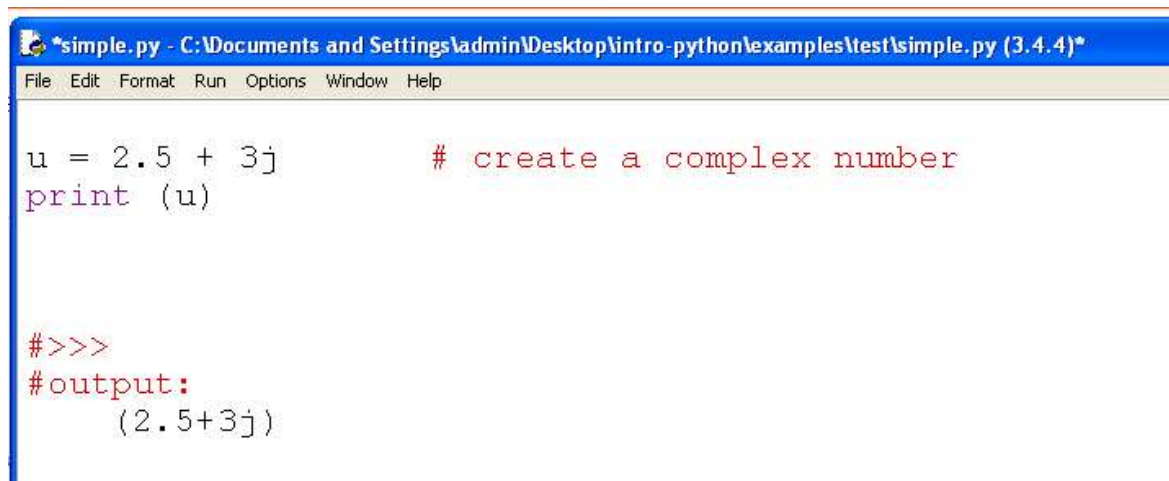
$$|u| = \sqrt{a^2 + b^2}$$

$$e^{iq} = \cos q + i \sin q$$

Complex Numbers

Python supports computation with complex numbers. The imaginary unit is written as `j` in Python, instead of i as in mathematics. A complex number $2 - 3i$ is therefore expressed as `(2-3j)` in Python. We remark that the number i is written as `1j`, not just `j`. Below is a sample session involving definition of complex numbers and some simple arithmetics:

Complex Numbers

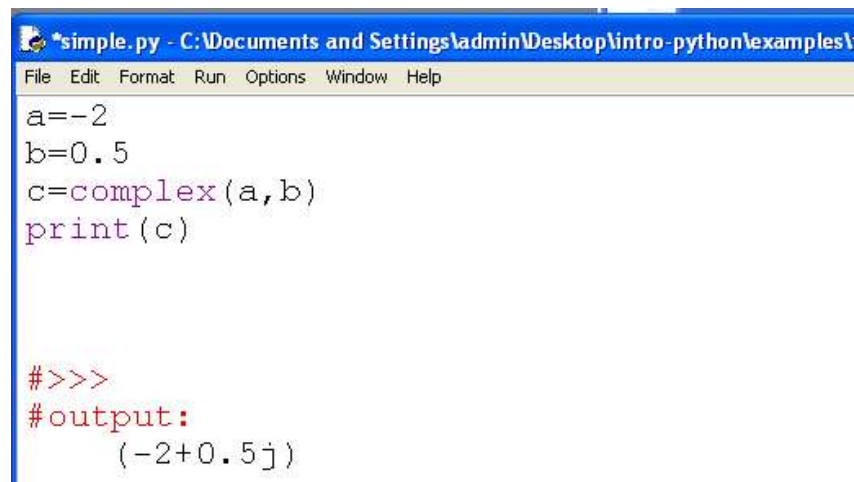


```
*simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)*
File Edit Format Run Options Window Help

u = 2.5 + 3j          # create a complex number
print (u)

#>>>
#output:
(2.5+3j)
```

Complex Numbers



A screenshot of a Python script window titled "simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The script content is as follows:

```
a=-2
b=0.5
c=complex(a,b)
print(c)
```

Below the script, the output is shown in red text:

```
#>>>
#output:
      (-2+0.5j)
```

Complex Numbers

add a complex number to a real power

simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)

File Edit Format Run Options Window Help

u = 2.5 + 3j # create a complex number

v=2

w=u+v

print("addition of u and v:", w)

x=u*w

print("multiplication of u and w:", x)

$u + v = (a + bi) + (c + di) = (a + c) + (b + d)i$

Python 3.4.4 Shell

File Edit Shell Debug Options Window Help

Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:tel)] on win32

Type "copyright", "credits" or "license()" for more

>>>

RESTART: C:\Documents and Settings\admin\Desktop\simple.py

addition of u and v: (4.5+3j)

multiplication of u and w: (2.25+21j)

Complex Numbers

simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)

File Edit Format Run Options Window Help

```
u = 2.5 + 3j          # create a complex number
```

```
v=2
```

```
w=u+v
```

```
x=u*w
```

```
print("real part is: ", x.real)
```

```
print("imaginary part is: ", x.imag)
```

```
print("conjugate of x is:", x.conjugate())
```

```
#>>>
```

```
#output:
```

```
real part is: 2.25
```

```
imaginary part is: 21.0
```

```
conjugate of x is: (2.25-21j)
```

Module for Number Type

- fraction
- math
- cmath
- random

- Using import statement

Fraction Type

- Python debuted a new numeric type, Fraction, which implements a rational number object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math.

Fraction Type

```
*frac.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/13-numbers/frac.py (3.4.4)*
File Edit Format Run Options Window Help
from fractions import Fraction
x = Fraction(1, 3)                # Numerator, denominator
y = Fraction(2, 6)                # Simplified to 2, 3 by gcd
print('x: ', x)
print('y: ', y)
z = x + y
print('Sum of x and y: ', z)

t = Fraction(.25) + Fraction('1.25') # 1/4 + 5/4
print('t: ', t)

#output

x:  1/3
y:  1/3
Sum of x and y:  2/3
t:  3/2
```


Math library

- This module is always available. It provides access to the mathematical functions defined by the C standard.

Math library

- `math.ceil(x)` :Return the ceiling of x, the smallest integer greater than or equal to x
- `math.copysign(x, y)`: Return a float with the magnitude (absolute value) of x but the sign of y. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns -1.0.
- `math.fabs(x)`: Return the absolute value of x.
- `math.factorial(x)`: Return x factorial
- `math.fmod(x, y)`: Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result

Math library

- `math.exp(x)`: Return e^x .
- `math.expm1(x)`: Return $e^x - 1$.
- `math.log(x[, base])`: With one argument, return the natural logarithm of x (to base e).
- `math.sqrt(x)`: Return the square root of x .

Math library

- `math.acos(x)`
- `math.asin(x)`
- `math.atan(x)`
- `math.atan2(y, x)`: Return `atan(y / x)`
- `math.cos(x)`
- `math.hypot(x, y)`: Return the Euclidean norm, `sqrt(x*x + y*y)`.
- `math.sin(x)`
- `math.tan(x)`

Math library

- `math.degrees(x)`: Convert angle x from radians to degrees.
- `math.radians(x)`: Convert angle x from degrees to radians.

Math library

- `math.acosh(x)`
- `math.asinh(x)`
- `math.atanh(x)`
- `math.cosh(x)`
- `math.sinh(x)`
- `math.tanh(x)`

Math library

- Constants

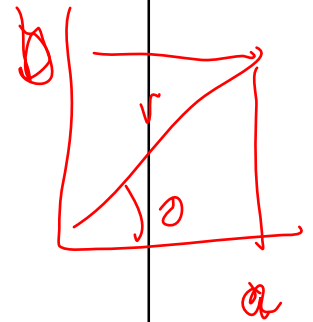
- `math.pi` : The mathematical constant $\pi = 3.141592\dots$, to available precision.
- `math.e` : The mathematical constant $e = 2.718281\dots$, to available precision.
- `math.inf`: A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.
- `math.nan`: A floating-point “not a number” (NaN) value. Equivalent to the output of `float('nan')`.

cmath library

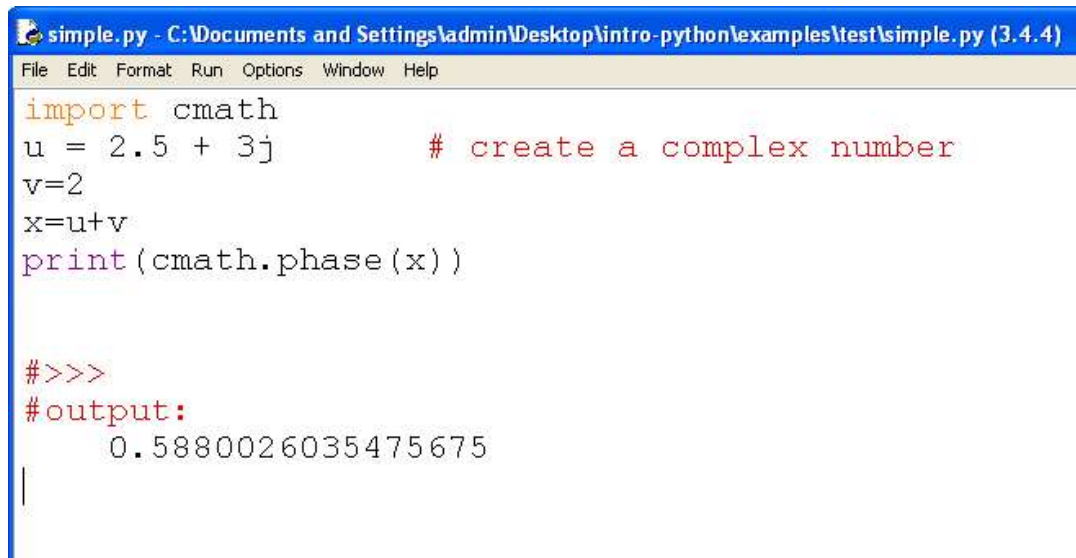
- This module is always available. It provides access to mathematical functions for complex numbers.

cmath: Conversions to and from polar coordinates

- `cmath.phase(x)` $x = a + bj$
- Return the phase of x (also known as the *argument* of x), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range $[-\pi, \pi]$, and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:



cmath: Conversions to and from polar coordinates

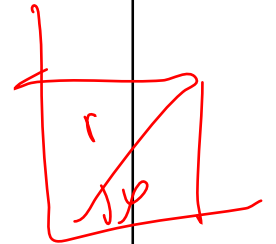


```
simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)
File Edit Format Run Options Window Help
import cmath
u = 2.5 + 3j          # create a complex number
v=2
x=u+v
print(cmath.phase(x))

#>>>
#output:
0.5880026035475675
|
```

cmath: Conversions to and from polar coordinates

- `cmath.polar(x)` : Return the representation of x in polar coordinates. Returns a pair (r, ϕ) where r is the modulus of x and ϕ is the phase of x . `polar(x)` is equivalent to `(abs(x), phase(x))`.
- `cmath.rect(r, ϕ)` : Return the complex number x with polar coordinates r and ϕ . Equivalent to $r * (\text{math.cos}(\phi) + \text{math.sin}(\phi)*1j)$.



cmath: Conversions to and from polar coordinates

```
*simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)*
File Edit Format Run Options Window Help
from cmath import *
x = 2.5 + 3j          # create a complex number

print(polar(x))
print(rect(3.905, 0.876))

#>>>
#output:
(3.905124837953327, 0.8760580505981934)
(2.5000942227048344+2.999758969917068j)
```

cmath: Power and logarithmic functions

- `cmath.exp(x)` Return the exponential value e^{**x} .
- `cmath.log(x[, base])` Returns the logarithm of x to the given *base*. If the *base* is not specified, returns the natural logarithm of x . There is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.
- `cmath.log10(x)` Return the base-10 logarithm of x . This has the same branch cut as `log()`.
- `cmath.sqrt(x)` Return the square root of x . This has the same branch cut as `log()`.

cmath: Trigonometric functions

- `cmath.acos(x)` Return the arc cosine of x . There are two branch cuts: One extends right from 1 along the real axis to ∞ , continuous from below. The other extends left from -1 along the real axis to $-\infty$, continuous from above.
- `cmath.asin(x)` Return the arc sine of x . This has the same branch cuts as `acos()`.
- `cmath.atan(x)` Return the arc tangent of x . There are two branch cuts: One extends from $1j$ along the imaginary axis to ∞j , continuous from the right. The other extends from $-1j$ along the imaginary axis to $-\infty j$, continuous from the left.
- `cmath.cos(x)` Return the cosine of x .
- `cmath.sin(x)` Return the sine of x .
- `cmath.tan(x)` Return the tangent of x .

cmath: Hyperbolic functions

- `cmath.acosh(x)` Return the inverse hyperbolic cosine of x . There is one branch cut, extending left from 1 along the real axis to $-\infty$, continuous from above.
- `cmath.asinh(x)` Return the inverse hyperbolic sine of x . There are two branch cuts: One extends from $1j$ along the imaginary axis to ∞j , continuous from the right. The other extends from $-1j$ along the imaginary axis to $-\infty j$, continuous from the left.
- `cmath.atanh(x)` Return the inverse hyperbolic tangent of x . There are two branch cuts: One extends from 1 along the real axis to ∞ , continuous from below. The other extends from -1 along the real axis to $-\infty$, continuous from above.
- `cmath.cosh(x)` Return the hyperbolic cosine of x .
- `cmath.sinh(x)` Return the hyperbolic sine of x .
- `cmath.tanh(x)` Return the hyperbolic tangent of x .

cmath: Classification functions

- `cmath.isfinite(x)` Return True if both the real and imaginary parts of `x` are finite, and False otherwise.
- New in version 3.2.
- `cmath.isinf(x)` Return True if either the real or the imaginary part of `x` is an infinity, and False otherwise.
- `cmath.isnan(x)` Return True if either the real or the imaginary part of `x` is a NaN, and False otherwise.

cmath: Constants

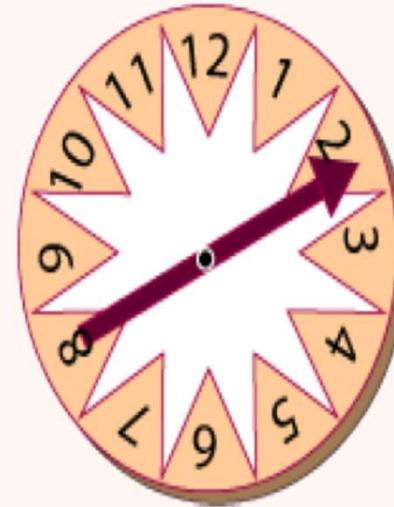
- `cmath.pi` The mathematical constant π , as a float.
- `cmath.e` The mathematical constant e , as a float.

Random Module

- This module implements pseudo-random number generators for various distributions.
- For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.
- On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Random Numbers

A single die can be regarded as a **random number generator**, where a single number is selected from a range (1 through 6) and each number is equally likely. The same is true for a second throw of the die. The outcome of the first throw does not affect the outcome of the second throw (or any other throw).



If you throw a die many times, you will get a sequence of random numbers. Since each number is equally likely, you would expect each number to occur about as often as any other number. For example, if you threw the die 60 times, you would expect to see about 10 of each number. However, it would be unusual to see exactly 10 of each number. The die does not "remember" its previous outcomes and does nothing to catch up with numbers that may have fallen short.

Random

Uniform Distribution

When each number in a range of numbers is equally likely, the numbers are **uniformly distributed**. Many random number generators, such as a single die or a spinner, produce a uniform distribution.

However, some random number generators are not uniform. For example, when two dice are thrown and the outcome is the sum of the spots, the range of possible outcomes is 2 through 12, but 7 is the most likely outcome since there are 6 combinations that sum to 7, more than any other number. Six and 8 are the second most likely outcomes, since there are 5 combinations each that sum to those numbers.

Sum of the Spots Shown
on Two Dice

		Die 1					
Die 2		1	2	3	4	5	6
	1	2	3	4	5	6	7
	2	3	4	5	6	7	8
	3	4	5	6	7	8	9
	4	5	6	7	8	9	10
	5	6	7	8	9	10	11
	6	7	8	9	10	11	12

Random Number Generator Algorithm

$$x_{n+1} = (a x_n + b) \bmod m$$

m is upper bound, x_0 is seed

$$0 < a, b, x_0 < m$$

a and b relatively prime

$$m=2^{32}, a=1664525, b=1013104223$$

$$x_0=13$$

$$\begin{aligned} x_0 &= 13 \\ x_1 &= a x_0 + b \bmod m \\ x_2 &= a x_1 + b \bmod m \\ a &= 2 \quad b = 3 \quad m = 7 \\ x_0 &= 1 \\ x_1 &= 2 + 3 = 5 \\ x_2 &= 10 + 3 = 13 \leq 6 \end{aligned}$$

Random Module

- Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937}-1$. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.
- The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

x5 3.0
1 756.432
18.3264

x5 3.0
756.432
18.3264

Random Module

- `random.random()` Return the next random floating point number in the range `[0.0, 1.0)`.
- `random.uniform(a, b)` Return a random floating point number N such that $a \leq N \leq b$ for $a \leq b$ and $b \leq N \leq a$ for $b < a$. The end-point value b may or may not be included in the range depending on floating-point rounding in the equation $a + (b-a) * \text{random}()$.
- `random.randint(a, b)`: Return a random integer N such that $a \leq N \leq b$

Random Module

- `random.seed(a=None, version=2)` Initialize the random number generator.
- If `a` is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the [os.urandom\(\)](#) function for details on availability).
- If `a` is an `int`, it is used directly.
- With version 2 (the default), a [str](#), [bytes](#), or [bytearray](#) object gets converted to an [int](#) and all of its bits are used. With version 1, the [hash\(\)](#) of `a` is used instead.
- Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

Random Module

- `random.gammavariate(alpha, beta)` Gamma distribution. (*Not* the gamma function!) Conditions on the parameters are $\alpha > 0$ and $\beta > 0$.
- The probability distribution function is:

$$\text{pdf}(x) = \frac{x^{(\alpha - 1)} * \text{math.exp}(-x / \beta)}{\text{math.gamma}(\alpha) * \beta^{\alpha}}$$

Random Module

- `random.triangular(low, high, mode)` Return a random floating point number N such that $low \leq N \leq high$ and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

Random Module

- `random.betavariate(alpha, beta)` Beta distribution. Conditions on the parameters are $\alpha > 0$ and $\beta > 0$. Returned values range between 0 and 1.
- `random.expovariate(lambd)` Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

Random Module

- `random.gauss(mu, sigma)` Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the [`normalvariate\(\)`](#) function defined below.
- `random.lognormvariate(mu, sigma)` Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.
- `random.normalvariate(mu, sigma)` Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.
- `random.vonmisesvariate(mu, kappa)` *mu* is the mean angle, expressed in radians between 0 and 2π , and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2π .

Random Module

- `random.paretovariate(alpha)` Pareto distribution. *alpha* is the shape parameter.
- `random.weibullvariate(alpha, beta)` Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.
- `class random.SystemRandom([seed])` Class that uses the [`os.urandom\(\)`](#) function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the [`seed\(\)`](#) method has no effect and is ignored. The [`getstate\(\)`](#) and [`setstate\(\)`](#) methods raise [`NotImplementedError`](#) if called.

Random Module

- `random.choice(seq)` Return a random element from the non-empty sequence `seq`. If `seq` is empty, raises [IndexError](#).
- `random.randrange(stop)` or `random.randrange(start, stop[, step])` Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.
- The positional argument pattern matches that of [range\(\)](#). Keyword arguments should not be used because the function may use them in unexpected ways.
- Changed in version 3.2: [randrange\(\)](#) is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

Random Module

- `random.shuffle(x[, random])` Shuffle the sequence `x` in place. The optional argument *random* is a 0-argument function returning a random float in `[0.0, 1.0)`; by default, this is the function `random()`.

Note that for even rather small `len(x)`, the total number of permutations of `x` is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

Random Module

- `random.sample(population, k)` Return a k length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

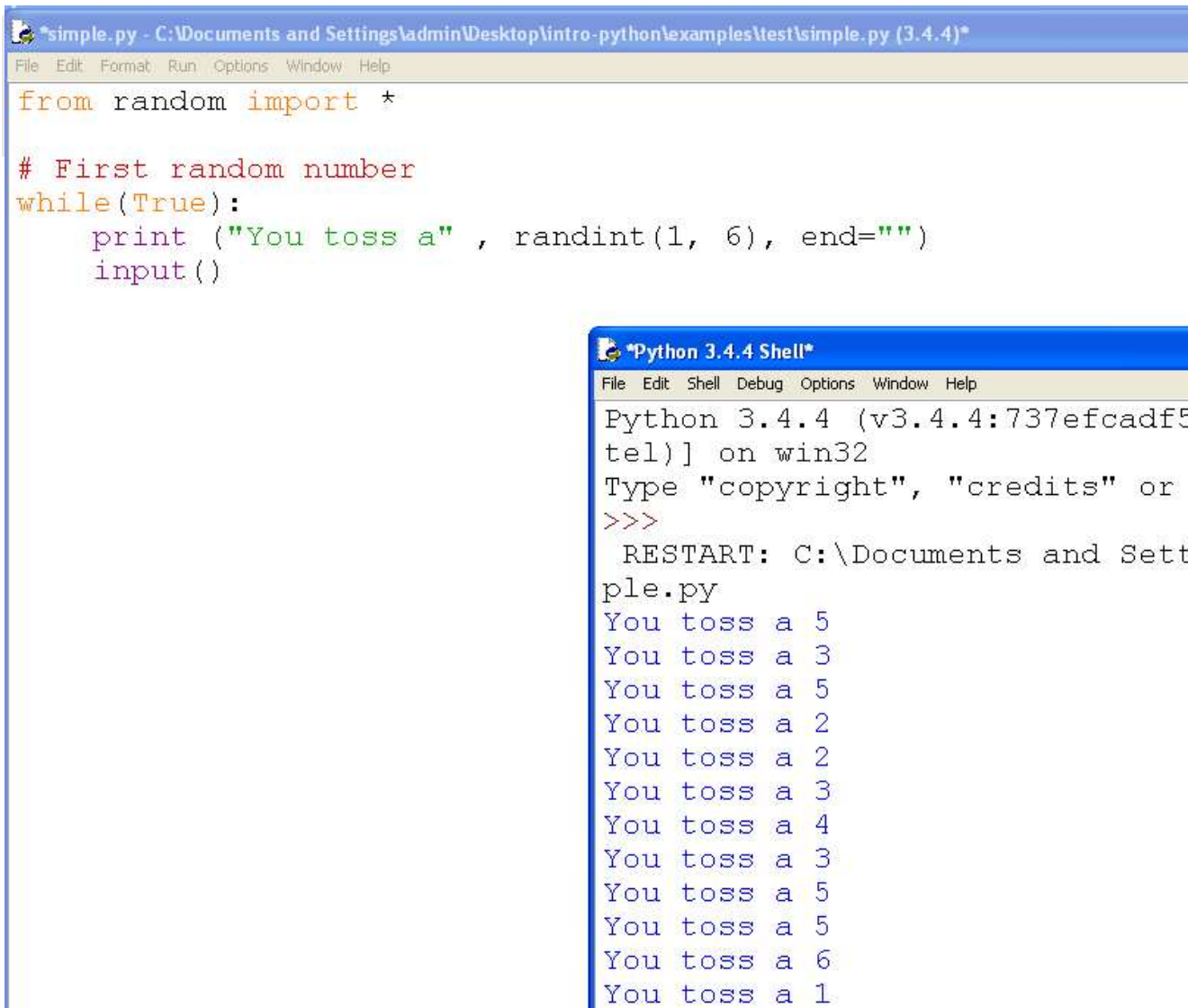
Members of the population need not be [hashable](#) or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use an [range\(\)](#) object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), 60)`.

If the sample size is larger than the population size, a [ValueError](#) is raised.

Random Module

- `random.getstate()` Return an object capturing the current internal state of the generator. This object can be passed to [`setstate\(\)`](#) to restore the state.
- `random.setstate(state)` *state* should have been obtained from a previous call to [`getstate\(\)`](#), and [`setstate\(\)`](#) restores the internal state of the generator to what it was at the time [`getstate\(\)`](#) was called.
- `random.getrandbits(k)` Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, [`getrandbits\(\)`](#) enables [`randrange\(\)`](#) to handle arbitrarily large ranges.



The image shows a Python IDE window titled `*simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)*`. The code in the editor is as follows:

```
from random import *  
  
# First random number  
while(True):  
    print ("You toss a" , randint(1, 6), end="")  
    input()
```

Below the IDE is a `*Python 3.4.4 Shell*` window. It displays the following output:

```
Python 3.4.4 (v3.4.4:737efcadf5  
tel)] on win32  
Type "copyright", "credits" or  
>>>  
RESTART: C:\Documents and Sett  
ple.py  
You toss a 5  
You toss a 3  
You toss a 5  
You toss a 2  
You toss a 2  
You toss a 3  
You toss a 4  
You toss a 3  
You toss a 5  
You toss a 5  
You toss a 6  
You toss a 1
```

Two-dice Gotcha!

In fact, if you use two random number generators, you need to be careful. The following code:

```
randint(1, 6)  
randint(1, 6)
```

will most likely produce two random number generators which are initialized to the same seed, and which will produce the same pseudorandom sequence. This is not what you want. The reason this happens is because `Random()` uses a seed based on the current time in milliseconds, but the current time changes little between the execution of the two statements. You could initialize the second random number generator using a random number from the first, but it is more convenient to use just one random number generator.

Throwing one die twice and adding up each outcome is equivalent to throwing two dice and adding up each die. However, throwing a 12-sided die once is not equivalent to throwing two 6-sided dice.

```
*simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)*
File Edit Format Run Options Window Help

# First random number
while(True):
    print ("You toss a" , randint(1, 6) + randint(1, 6) , end="")
    input()

#>>>
#output:
You toss a 12
You toss a 6
You toss a 8
You toss a 4
You toss a 5
You toss a 7
You toss a 6
You toss a 9
You toss a 7
You toss a 6
You toss a 9
You toss a 10
You toss a 2
You toss a 2
You toss a 8
You toss a 10
You toss a 4
You toss a 11
You toss a 10
You toss a 11
You toss a 9
```

Random

- We would like to write a program for calculating $\sin(x)$ for random angles (in radians) between -10π and $+10\pi$

simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)

File Edit Format Run Options Window Help

```
from random import *
from math import *
j=0
print ("\t x \t\t sin(x)")
print ("-----\t\t-----")
while(j<10):
    x=random()*(20*pi)-10*pi
    print (" ", x, "\t", sin(x))
    j=j+1
```

x	sin(x)
15.525472259918288	0.18147977798114912
30.87010177063307	-0.5191231933872631
19.338401896931266	0.46960734118298914
-4.328292703231188	0.9271374555340182
28.562038965690334	-0.283752384237605
14.001975254161167	0.9908755137879238
2.6488302636800967	0.4730614368325556
-7.506028816537135	-0.9400727183348745
-27.87021586247229	-0.3932079771413325
-15.051176121592164	-0.6105755624962561

Random

mult.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\20-1\mult.py (3.4.4)

File Edit Format Run Options Window Help

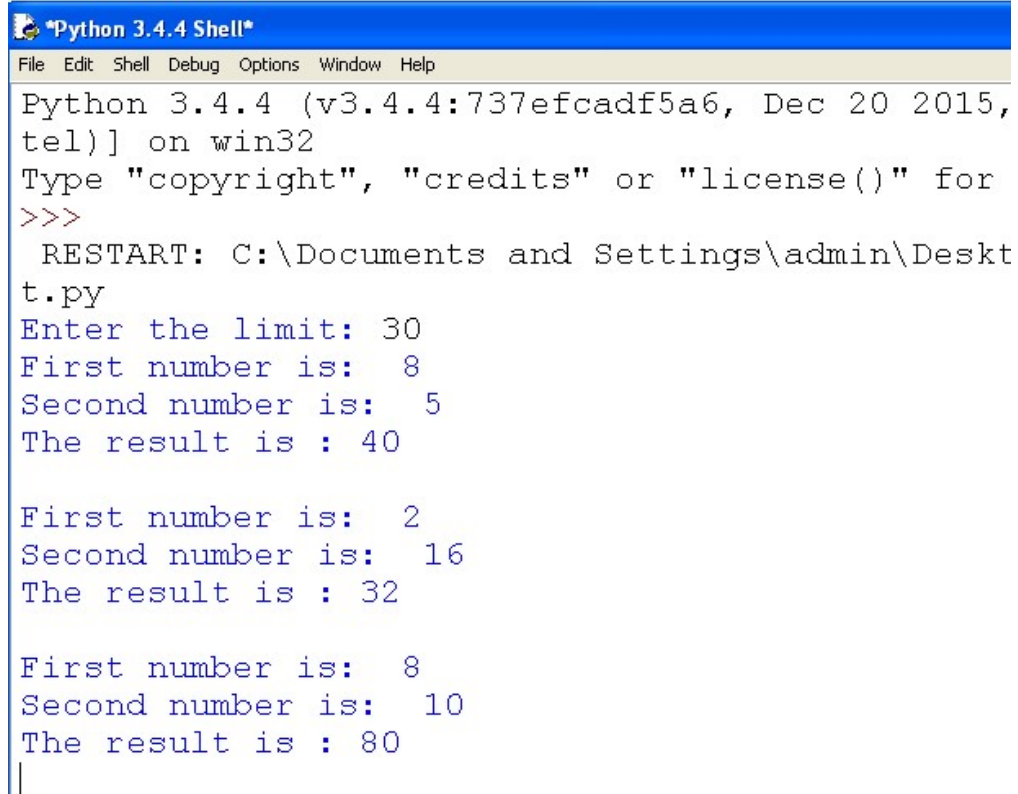
```
from random import *

def multiply(A,B):
    C= 0
    while (A != 0) :
        if ((A % 2)==1) : C=C+B
        A = A // 2
        B = B * 2
    return(C)

limit=int(input("Enter the limit: "))
while(True):
    a=randint(1, limit)
    b=randint(1, limit)
    print("First number is: ", a)
    print("Second number is: ", b)
    result=multiply(a,b)
    print("The result is :", result) ;
    input()
```

A	B	C
9	7	0
4	14	7
2	28	
1	56	63
0	112	<u>63</u>

Random



```
*Python 3.4.4 Shell*
File Edit Shell Debug Options Window Help
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015,
tel)] on win32
Type "copyright", "credits" or "license()" for
>>>
  RESTART: C:\Documents and Settings\admin\Deskt
t.py
Enter the limit: 30
First number is: 8
Second number is: 5
The result is : 40

First number is: 2
Second number is: 16
The result is : 32

First number is: 8
Second number is: 10
The result is : 80
|
```



```
simple.py - C:\Documents and Settings\admin\Desktop\intro-python\examples\test\simple.py (3.4.4)
File Edit Format Run Options Window Help

import random

random.seed( 10 )
print ("Random number with seed 10 : ", random.random())

# It will generate same random number
random.seed( 10 )
print ("Random number with seed 10 : ", random.random())

# It will generate same random number
random.seed( 10 )
print ("Random number with seed 10 : ", random.random())
```

```
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help

Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:2
tel)] on win32
Type "copyright", "credits" or "license()" for more
>>>
RESTART: C:\Documents and Settings\admin\Desktop\ir
ple.py
Random number with seed 10 : 0.5714025946899135
Random number with seed 10 : 0.5714025946899135
Random number with seed 10 : 0.5714025946899135
```

Shared References

num-simple.py - /home/nowzari/Desktop/python/pytho
File Edit Format Run Options Window Help

```
X=42  
Y=42  
print(X==Y)  
print(X is Y)
```

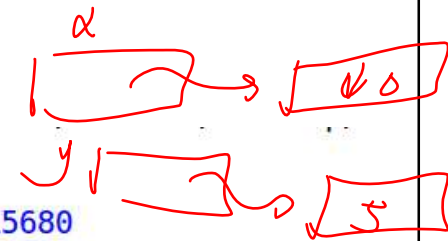
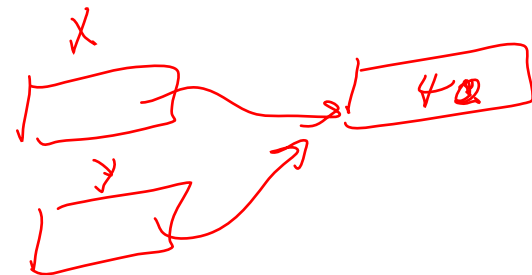
```
print("id X is", id(X))  
print("id Y is", id(Y))
```

```
X=40  
Y=X  
print(X==Y)  
print(X is Y)
```

```
print("id X is", id(X))  
print("id Y is", id(Y))
```

```
X=40  
Y=5  
print(X==Y)  
print(X is Y)
```

```
print("id X is", id(X))  
print("id Y is", id(Y))
```



```
True  
True  
id X is 10915680  
id Y is 10915680  
True  
True  
id X is 10915616  
id Y is 10915616  
False  
False  
id X is 10915616  
id Y is 10914496
```

Parameter passing

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.

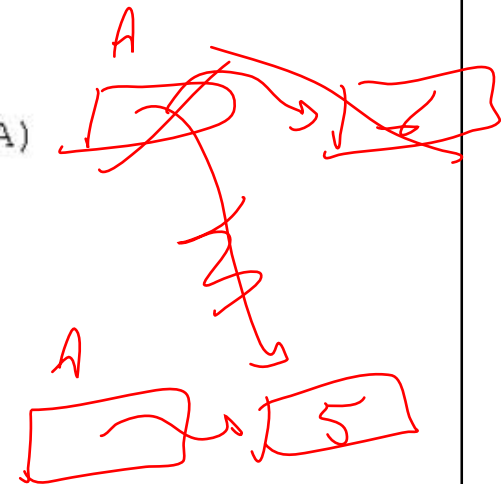
Parameter passing

```
*num-par.py - C:/Documents and Settings/admin/Desktop/intro-python/examples-2/12-numbers/num-par
File Edit Format Run Options Window Help

def func(A):
    A=A+1
    print("A inside the function: ", A)

A=5
print("A Befor function call: ", A)
func(A)
print("A After function call: ", A)

#output
A Befor function call: 5
A inside the function: 6
A After function call: 5
```



End