Homomorphic Processor Documentation

Release 0.1

Louis GERARD and Edouard SAMYN

Apr 26, 2016

1 homomorphic-processor **3**
1.1 datatypes package . . . 3
1.1.1 Subpackages . . . 3
datatypes.bits package . . . 3
datatypes.integers package . . . 7
1.2 demo package . . . 15
1.2.1 Submodules . . . 15
1.2.2 demo.ImageUtils module . . . 15
1.3 main module . . . 16
1.4 testing package . . . 16
1.4.1 Submodules . . . 16
1.4.2 testing.TestBitsMethods module . . . 16
1.4.3 testing.TestInt8Methods module . . . 17
1.4.4 testing.TestNoise module . . . 17
1.4.5 testing.TestUInt8Methods module . . . 17
1.4.6 testing.TestUIntMethods module . . . 18

Python Module Index **19**

Index **21**

Contents:

CHAPTER

ONE

HOMOMORPHIC-PROCESSOR

1.1 datatypes package

1.1.1 Subpackages

datatypes.bits package

Subpackages

datatypes.bits.errors package

Submodules

datatypes.bits.errors.BitwiseOperationError module

exception datatypes.bits.errors.BitwiseOperationError.**BitwiseOperationError**(*message*)

Bases: exceptions.RuntimeError

datatypes.bits.errors.InstantiationError module

exception datatypes.bits.errors.InstantiationError.**InstantiationError**(*message*)

Bases: exceptions.RuntimeError

datatypes.bits.errors.OpNotAllowedError module

exception datatypes.bits.errors.OpNotAllowedError.**OpNotAllowedError**(*message*)

Bases: exceptions.RuntimeError

Submodules

datatypes.bits.Bit module

class datatypes.bits.Bit.**Bit**

General purpose bit, might be encrypted or not. This is an abstract class and therefore might not be instantiated

Raises **InstantiationError** – if called.

AND(*other*)

This method should be overridden in subclasses to perform a logical AND operation between two bits.

Raises **OpNotAllowedError** – if called

NOT()
This method should be overridden in subclasses to perform a logical NOT operation on self.

Raises **OpNotAllowedError** – if called

OR(*other*)
This method should be overridden in subclasses to perform a logical OR operation between two bits.

Raises **OpNotAllowedError** – if called

XOR(*other*)
This method should be overridden in subclasses to perform a logical XOR operation between two bits.

Raises **OpNotAllowedError** – if called

__add__(*other*)
This method just calls the OR operation We overrode this operator to be able to write logic equations more easily

__and__(*other*)
This method just calls the AND operation We overrode this operator to be able to write logic equations more easily

__invert__()
This method just calls the NOT operation We overrode this operator to be able to write logic equations more easily

__mul__(*other*)
This method just calls the AND operation We overrode this operator to be able to write logic equations more easily

__neg__()
This method just calls the NOT operation We overrode this operator to be able to write logic equations more easily

__or__(*other*)
This method just calls the OR operation We overrode this operator to be able to write logic equations more easily

__xor__(*other*)
This method just calls the XOR operation We overrode this operator to be able to write logic equations more easily

debug__printAsBoolean()
This method is for debug purposes only, it will reveal the Boolean value in the bit

Raises **OpNotAllowedError** – if called

class datatypes.bits.Bit.**CryptoBit**(*value=False*, *verbose=False*)
Bases: *datatypes.bits.Bit.Bit*

Emulated homomorphic bit, we forbid ourselves to read the bit value. Upon creation, the apparent value of the bit will be random. The plain value is stored in the __bit member.

:param value:(optional) Defines the value of the bit, defaults to False :param verbose:(optional) Defines whether or not the bit will print a message upon refresh, defaults to False

AND(*other*)
Performs a logical AND operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the AND operation

Return type *CryptoBit*

Raises **BitwiseOperationError** – if the right operand is not a Bit

NOT()
Performs a logical NOT operation on self

Returns A bit containing the result of the NOT operation

Return type *CryptoBit*

OR(*other*)
Performs a logical OR operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the OR operation

Return type *CryptoBit*

Raises **BitwiseOperationError** – if the right operand is not a Bit

XOR(*other*)
Performs a logical XOR operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the XOR operation

Return type *CryptoBit*

Raises **BitwiseOperationError** – if the right operand is not a Bit

__eq__(*other*)
Test if two bits are Equals, however since two Cryptobits cannot be compared directly, this function always returns an exception

Raises **OpNotAllowedError** – if other is not of type PlainBit

__repr__()
Return representation of Crypto

Note that this value is not the real value but a random value, so you should not rely on it :returns: 1 if bit is set to True, 0 otherwise

debug__printAsBoolean()
This method is for debug purposes only, it will reveal the Boolean value in the bit

Returns Boolean value of bit

Return type Boolean

decrypt()
Decrypt the CryptoBit

Returns PlainBit containing the value of the CryptoBit

Return type PlainBit

refresh()
Resets the bit's noise, if the bit's verbose attribute is set, then it will print a message

setNoise(*value=0*)
Sets the bit's noise, if the noise is above the threshold, then it will trigger a refresh

Parameters **value** (*int*) – The value that should be set, defaults to 0

class datatypes.bits.Bit.**PlainBit**(*value=False*)

Bases: *datatypes.bits.Bit.Bit*

This class represents a bit which is not encrypted Its goal is just to wrap a boolean and allow us to do operation with CryptoBits

Parameters **value** (*Boolean*) – Defines the value of the bit, defaults to False

AND(*other*)

Performs a logical AND operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the AND operation

Return type PlainBit if right operand is PlainBit, CryptoBit if right operand is CryptoBit

Raises **BitwiseOperationError** – if the right operand is not a Bit

NOT()

Performs a logical NOT operation on self

Returns A bit containing the result of the NOT operation

Return type PlainBit

OR(*other*)

Performs a logical OR operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the OR operation

Return type PlainBit if right operand is PlainBit, CryptoBit if right operand is CryptoBit

Raises **BitwiseOperationError** – if the right operand is not a Bit

XOR(*other*)

Performs a logical XOR operation between self and other

Parameters **other** (*PlainBit or CryptoBit*) – RightOperand

Returns A bit containing the result of the XOR operation

Return type PlainBit if right operand is PlainBit, CryptoBit if right operand is CryptoBit

Raises **BitwiseOperationError** – if the right operand is not a Bit

__eq__(*other*)

Test if two bits are Equals

Parameters **other** (*PlainBit*) – Bit to compare

Returns True if bits are equal, False otherwise

Return type Boolean

Raises **OpNotAllowedError** – if other is not of type PlainBit

__repr__()

Return representation of Plain Bit

Returns 1 if bit is set to True, 0 otherwise

debug__printAsBoolean()

This method is for debug purposes only, it will reveal the Boolean value in the bit

Returns Boolean value of bit

Return type Boolean

encrypt()
Encrypts the PlainBit

Returns CryptoBit containing the value of the PlainBit

Return type *CryptoBit*

datatypes.integers package

Subpackages

datatypes.integers.error package

Submodules

datatypes.integers.error.BadRightOperand module

exception datatypes.integers.error.BadRightOperand.**BadRightOperand**
Bases: exceptions.RuntimeError

datatypes.integers.error.OverflowError module

exception datatypes.integers.error.OverflowError.**OverflowError**(*message*)
Bases: exceptions.RuntimeError

Submodules

datatypes.integers.Int8 module

class datatypes.integers.Int8.**Int8**(*value=0*, *bits=None*, *randomize=False*)
Signed integer made from an array of 8 Bits. (from -128 to 127) the representation is in two's complement We chose a convention for our arrays of bits : when we initiate a byte, the first bit of the array is the Most Significant Bit. The last one is the Least Significant Bit. Thus, bits[0] = MSB, bits[bits.length()] = LSB

Raises **InstantiationError** – if the array is too long/short

__abs__()
Returns the absolute value. Since our integers are coded in two's complement, absolute value of -128 cannot be represented. However if self equals -128 the result will have its testOverflow bit set to indicate that an overflowError has occurred.

Returns the absolute value, except for -128 for which it returns 0.

Return type *Int8*

__add__(*other*)
This method will add 2 Int8. We override this operator to be able to write operations more easily

Parameters **other** (Int8) – Right Operand

Returns the sum of the two Int8

Return type *Int8*

__and__(*other*)
bitwise AND operation, returns an Int8 We override this operator to be able to write operations more easily

Parameters **other** (Int8) – Right Operand

Returns bitwise AND operation, returns an Int8

Return type *Int8*

__div__(*other*)
This method will divide 2 Int8. We override this operator to be able to write operations more easily DIVISION IS ROUNDED TOWARDS 0 since it uses the division from the absolutes value of the arguments given. This means that there might be an isssue when using -128 as an operand, cf __abs__ function

Parameters **other** (Int8) – Right Operand

Returns the result of division of the two Int8

Return type *Int8*

__eq__(*other*)
test if two Int8 are equal

Parameters **other** (Int8) – Int8 to compare

Returns True or False, in form of a CryptoBit or a PlainBit

Return type CryptoBit or PlainBit

__ge__(*other*)
test if an Int8 is greater or equal than/to another one

Parameters **other** (Int8) – Int8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__gt__(*other*)
test if an Int8 is greater than another one

Parameters **other** (Int8) – Int8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__invert__()
returns the one's complement Int8 We override this operator to be able to write operations more easily

Returns the one's complement Int8

Return type *Int8*

__le__(*other*)
test if an Int8 is lesser or equal than/to another one

Parameters **other** (Int8) – Int8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__len__()
returns the length of the array of bits of an Int8 (which is always 8) We override this operator to be able to write operations more easily

Returns an integer, the size of the array of bits of Int8 (which is 8)

Return type integer

__lshift__(*other*)
returns an Int8 that its bits were shifted to the left We override this operator to be able to write operations more easily Note that this operation performs an arithmetic shift and not a binary shift This means that the sign bit is ignored by the operation, and it will not be shifted as a consequence

Parameters **other** (*integer*) – integer to know how much we shift

Returns an Int8, its bits were shifted to the left

Return type *Int8*

__lt__(*other*)
test if an Int8 is lesser than another one

Parameters **other** (Int8) – Int8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__mod__(*other*)
This method will return the remainder of the division between 2 Int8. We override this operator to be able to write operations more easily As a convention the result will be positive and will be the rest from the euclidian division of the absolute values of the two arguments

Parameters **other** (Int8) – Right Operand

Returns the remainder of the division between two Int8

Return type *Int8*

__neg__()
returns the opposite Int8 We override this operator to be able to write operations more easily

Returns opposite Int8, except for -128

Return type *Int8*

__or__(*other*)
bitwise OR operation, returns an Int8 We override this operator to be able to write operations more easily

Parameters **other** (Int8) – Right Operand

Returns bitwise OR operation, returns an Int8

Return type *Int8*

__repr__()
Return representation of Int8

Returns integer, the value of the Int8

Return type integer

__rshift__(*other*)
returns an Int8 that its bits were shifted to the right We override this operator to be able to write operations more easily Note that this operation performs an arithmetic shift and not a binary shift This means that the sign bit is ignored by the operation, and it will not be shifted as a consequence

Parameters **other** (*integer*) – integer to know how much we shift

Returns an Int8, its bits were shifted to the right

Return type *Int8*

__sub__(*other*)
This method will substract 2 Int8. We override this operator to be able to write operations more easily

Parameters **other** (Int8) – Right Operand

Returns the difference of the two Int8

Return type *Int8*

__xor__(*other*)
bitwise XOR operation, returns an Int8 We override this operator to be able to write operations more easily

Parameters **other** (Int8) – Right Operand

Returns bitwise XOR operation, returns an Int8

Return type *Int8*

debug_showValue()
This method is for debug purposes only, it will reveal the integer value of the Int8

Returns an integer, the value of the Int8

Return type integer

decrypt()
This method makes every CryptoBit into PlainBit

encrypt()
This method makes every PlainBit into CryptoBit

toUInt8()
returns the abs value of integer as Unisgned Integer (UInt8)

Returns abs value of integer as UInt8

Return type *UInt8*

datatypes.integers.UInt module

class datatypes.integers.UInt.**UInt**(*value=0*, *ints=None*, *fixedSize=None*, *randomize=False*)
Unsigned integer made from an array of UInt8s. We chose a convention for our arrays of bits : when we initiate a byte, the first bit of the array is the Most Significant Bit. The last one is the Least Significant Bit.

Raises **InstantiationError** – if the array is too long/short

__abs__()
return the absolute value which is itself

Returns the absolute value (itself)

Return type *UInt*

__add__(*other*)
We wanted to append another 8-bit word if an overflow was detected, however we realized it could weaken the encryption. Therefore, we decided to return both the incorrect result and the carryOut bit to be properly handled by the user once decrypted. There may be some error in this function. When we add two UInt of different sizes, or when we add an UInt of value zero.

/!Be aware of this potential error

Parameters **other** – UInt

Returns The sum of two UInts

Return type *UInt*

__eq__(*other*)
test if two UInts are equal

Parameters **other** (UInt) – UInt to compare

Returns True or False, in form of a CryptoBit or a PlainBit

Return type CryptoBit or PlainBit

Raises BadRightOperand

__len__()
returns the length of the array of UInt8s We override this operator to be able to write operations more easily

Returns an integer, the size of the array of UInt8s

Return type integer

__sub__(*other*)
This method will subtract 2 UInt. We override this operator to be able to write operations more easily

Parameters **other** (UInt) – Right Operand

Returns the difference of the two UInt

Return type *UInt*

debug_showValue()
This method is for debug purposes only, it will reveal the integer value of the UInt

Returns an integer, the value of the UInt

Return type integer

datatypes.integers.UInt8 module

class datatypes.integers.UInt8.**UInt8**(*value=0*, *bits=None*, *randomize=False*)
Unsigned integer made from an array of 8 Bits. (from 0 to 255) We chose a convention for our arrays of bits : when we initiate a byte, the first bit of the array is the Most Significant Bit. The last one is the Least Significant Bit. Thus, bits[0] = MSB, bits[bits.length()] = LSB Please note that this class can also be used to represent characters for example. Since we cannot access the data, its meaning is irrelevant to us, therefore UInt8 might as well be considered as Char

Raises **InstantiationError** – if the array is too long/short

__abs__()
return the absolute value which is itself

Returns the absolute value (itself)

Return type *UInt8*

__add__(*other*)
This method will add 2 UInt8. We override this operator to be able to write operations more easily

Parameters **other** (UInt8) – Right Operand

Returns the sum of the two UInt8

Return type *UInt8*

__and__(*other*)
bitwise AND operation, returns an UInt8 We override this operator to be able to write operations more easily

Parameters **other** (UInt8) – Right Operand

Returns bitwise AND operation, returns an UInt8

Return type *UInt8*

__div__(*other*)
This method will divide 2 UInt8. We override this operator to be able to write operations more easily Note that by convention, the result will always be rounded towards 0

Parameters **other** (UInt8) – Right Operand

Returns the result of division of the two UInt8

Return type *UInt8*

__eq__(*other*)
test if two UInt8 are equal

Parameters **other** (UInt8) – UInt8 to compare

Returns True or False, in form of a CryptoBit or a PlainBit

Return type CryptoBit or PlainBit

__ge__(*other*)
test if an UInt8 is greater or equal than/to another one

Parameters **other** (UInt8) – UInt8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__gt__(*other*)
test if an UInt8 is greater than another one

Parameters **other** (UInt8) – UInt8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__invert__()
returns the one's complement UInt8 We override this operator to be able to write operations more easily

Returns the one's complement UInt8

Return type *UInt8*

__le__(*other*)
test if an UInt8 is lesser or equal than/to another one

Parameters **other** (UInt8) – UInt8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__len__()
returns the length of the array of bits of an UInt8 (which is always 8) We override this operator to be able to write operations more easily

Returns an integer, the size of the array of bits of UInt8 (which is 8)

Return type integer

__lshift__(*other*)
returns an UInt8 that its bits were shifted to the left We override this operator to be able to write operations more easily Note that this operation performs an arithmetic shift and not a binary shift

Parameters **other** (*integer*) – integer to know how much we shift

Returns an UInt8, its bits were shifted to the left

Return type *UInt8*

__lt__(*other*)
test if an UInt8 is lesser than another one

Parameters **other** (UInt8) – UInt8 to compare

Returns CryptoBit or PlainBit (true or false)

Return type CryptoBit or PlainBit

__mod__(*other*)
This method will return the remainder of the division between 2 UInt8. We override this operator to be able to write operations more easily Note that by convention : The result will always be positive

Parameters **other** (UInt8) – Right Operand

Returns the remainder of the division between two UInt8

Return type *UInt8*

__mul__(*other*)
This method will multiply 2 UInt8. We override this operator to be able to write operations more easily The multiplication seems to work fine, however there may be a problem in the addition of two UInt Since the addition od two UInt is used in this method there may be some error see addition of UInts of different sizes.

Parameters **other** (UInt8) – Right Operand

Returns the product of the two UInt8 which is an UInt

Return type *UInt*

__neg__()
raise RuntimeError

Raises RuntimeErro

__or__(*other*)
bitwise OR operation, returns an UInt8 We override this operator to be able to write operations more easily

Parameters **other** (UInt8) – Right Operand

Returns bitwise OR operation, returns an UInt8

Return type *UInt8*

__repr__()
Return representation of UInt8

Returns integer, the value of the UInt8

Return type integer

__rshift__(*other*)
returns an UInt8 that its bits were shifted to the right We override this operator to be able to write operations more easily Note that this operation performs an arithmetic shift and not a binary shift, this means that the new bit inserted will be equal to the MSB

Parameters **other** (*integer*) – integer to know how much we shift

Returns an UInt8, its bits were shifted to the right

Return type *UInt8*

__sub__(*other*)
This method will subtract 2 UInt8. We override this operator to be able to write operations more easily

Parameters **other** (UInt8) – Right Operand

Returns the difference of the two UInt8

Return type *UInt8*

__xor__(*other*)
bitwise XOR operation, returns an UInt8 We override this operator to be able to write operations more easily

Parameters **other** (UInt8) – Right Operand

Returns bitwise XOR operation, returns an UInt8

Return type *UInt8*

debug_showValue()
This method is for debug purposes only, it will reveal the integer value of the UInt8

Returns an integer, the value of the UInt8

Return type integer

decrypt()
This method makes every CryptoBit into PlainBit

encrypt()
This method makes every PlainBit into CryptoBit

showValue()
This method will show the 'fake' value. It will read the value of the CryptoBit

Returns an integer, the 'fake' value of the UInt8

Return type integer

toInt8()
returns the value of integer as Signed Integer (Int8) overflow might occur, if this is the case, the variable testOverflow of the returned Integer will be set to true

Returns value of integer as Int8

Return type *Int8*

datatypes.integers.Utility module

datatypes.integers.Utility.**completeAddOnOneBit**(*firstBit*, *secondBit*, *carry*)
This method will add 2 Bits.

Parameters

- **firstBit** (*Bit (CryptoBit or PlainBit)*) – Left Operand
- **secondBit** (*Bit (CryptoBit or PlainBit)*) – Right Operand

- **carry** (*Bit (CryptoBit or PlainBit)*) – The carry for the addition. A Bit (CryptoBit or PlainBit)

Returns A bit : the result of XOR between the two Bits. And a Bit that is the carry

Return type Bit, Bit

datatypes.integers.Utility.**completeSubOnOneBit**(*minuend*, *subtrahend*, *borrowIn*)
This method will subtract 2 Bits.

Parameters

- **minuend** (*Bit (CryptoBit or PlainBit)*) – Left Operand
- **subtrahend** (*Bit (CryptoBit or PlainBit)*) – Right Operand
- **borrowIn** (*Bit (CryptoBit or PlainBit)*) – The borrowIn for the subtraction. A Bit (CryptoBit or PlainBit)

Returns A bit : the result of XOR between the two Bits. And a Bit that is the BorrowIn

Return type Bit, Bit

1.2 demo package

1.2.1 Submodules

1.2.2 demo.ImageUtils module

demo.ImageUtils.**decode**(*pixels*, *path*)
This function will take a 2D Matrix of 3-sized tuple UInt8 and return the corresponding image

Parameters

- **pixels** (*2D Matrix of UInt8 tuple of size 3*) – This is the pixels data
- **path** (*String*) – This is the path that the image will be saved on

Returns Pixels Matrix encoded with our datatypes

Return type 2D Matrix of UInt8 tuple of size 3

demo.ImageUtils.**decrypt**(*pixels*)
This function will take an encrypted 2D Matrix of 3-sized tuple UInt8 and decrypt it

Parameters **pixels** (*2D Matrix of UInt8 tuple of size 3*) – This is the pixels data

Returns Decrypted Pixels Matrix encoded with our datatypes

Return type 2D Matrix of UInt8 tuple of size 3

demo.ImageUtils.**dump**(*data*, *path*)
This will write a python object to a file, in our case we will use it to transfer the data between bob and alice

Parameters

- **data** (*Python Object*) – The python object to save
- **path** (*String*) – Path to save the object

demo.ImageUtils.**encode**(*path*)
This function will take an RGB image of bit depth 24 and encode all of its data into our datatypes

Parameters **path** (*String*) – This is the path to the image, image must be of type RGB and and a 24 bit depth

Returns Pixels Matrix encoded with our datatypes

Return type 2D Matrix of UInt8 tuple of size 3

demo.ImageUtils.**encrypt**(*pixels*)
This function will take a 2D Matrix of 3-sized tuple UInt8 and encrypt it

Parameters **pixels** (*2D Matrix of UInt8 tuple of size 3*) – This is the pixels data

Returns Encrypted Pixels Matrix encoded with our datatypes

Return type 2D Matrix of UInt8 tuple of size 3

demo.ImageUtils.**load**(*path*)
This function will read a python object written with pickel from a file

Parameters **path** (*String*) – Path to file to read

Returns Python object retrieved from the file

Return type Python Object

demo.ImageUtils.**negate**(*pixels*)
This function will take a 2D Matrix of 3-sized tuple UInt8 and invert its values

Parameters **pixels** (*2D Matrix of UInt8 tuple of size 3*) – This is the pixels data

Returns Negated Pixels Matrix

Return type 2D Matrix of UInt8 tuple of size 3

1.3 main module

main.**result = <unittest.runner.TextTestResult run=31 errors=0 failures=3>**
int1 = UInt8(value=54) int2 = UInt8(value=19)

int3 = int1 * int2

int3.debug_showValue()

1.4 testing package

1.4.1 Submodules

1.4.2 testing.TestBitsMethods module

class testing.TestBitsMethods.**TestBitsMethods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_and()

test_or()

test_xor()

1.4.3 testing.TestInt8Methods module

class testing.TestInt8Methods.**TestInt8Methods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

test_int8_abs()

test_int8_addition()

test_int8_debugshowValue()

test_int8_division()

test_int8_eq_ne()

test_int8_ge()

test_int8_gt()

test_int8_le()

test_int8_lshift()

test_int8_lt()

test_int8_mod()

test_int8_rshift()

test_int8_subtraction()

1.4.4 testing.TestNoise module

1.4.5 testing.TestUInt8Methods module

class testing.TestUInt8Methods.**TestUInt8Methods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

test_uint8_addition()

test_uint8_debugshowValue()

test_uint8_division()

test_uint8_eq_ne()

test_uint8_ge()

test_uint8_gt()

test_uint8_le()

test_uint8_lshift()

test_uint8_lt()

test_uint8_mod()

test_uint8_rshift()

test_uint8_subtraction()

1.4.6 testing.TestUIntMethods module

class testing.TestUIntMethods.**TestUIntMethods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

test_uint_addition()

test_uint_debugshowValue()

test_uint_subtraction()

d

datatypes.bits.Bit, 3
datatypes.bits.errors.BitwiseOperationError, 3
datatypes.bits.errors.InstantiationError, 3
datatypes.bits.errors.OpNotAllowedError, 3
datatypes.integers.error.BadRightOperand, 7
datatypes.integers.error.OverflowError, 7
datatypes.integers.Int8, 7
datatypes.integers.UInt, 10
datatypes.integers.UInt8, 11
datatypes.integers.Utility, 14
demo.ImageUtils, 15

m

main, 16

t

testing.TestBitsMethods, 16
testing.TestInt8Methods, 17
testing.TestUInt8Methods, 17
testing.TestUIntMethods, 18

Symbols

__abs__() (datatypes.integers.Int8.Int8 method), 7
__abs__() (datatypes.integers.UInt.UInt method), 10
__abs__() (datatypes.integers.UInt8.UInt8 method), 11
__add__() (datatypes.bits.Bit.Bit method), 4
__add__() (datatypes.integers.Int8.Int8 method), 7
__add__() (datatypes.integers.UInt.UInt method), 10
__add__() (datatypes.integers.UInt8.UInt8 method), 11
__and__() (datatypes.bits.Bit.Bit method), 4
__and__() (datatypes.integers.Int8.Int8 method), 7
__and__() (datatypes.integers.UInt8.UInt8 method), 11
__div__() (datatypes.integers.Int8.Int8 method), 8
__div__() (datatypes.integers.UInt8.UInt8 method), 12
__eq__() (datatypes.bits.Bit.CryptoBit method), 5
__eq__() (datatypes.bits.Bit.PlainBit method), 6
__eq__() (datatypes.integers.Int8.Int8 method), 8
__eq__() (datatypes.integers.UInt.UInt method), 10
__eq__() (datatypes.integers.UInt8.UInt8 method), 12
__ge__() (datatypes.integers.Int8.Int8 method), 8
__ge__() (datatypes.integers.UInt8.UInt8 method), 12
__gt__() (datatypes.integers.Int8.Int8 method), 8
__gt__() (datatypes.integers.UInt8.UInt8 method), 12
__invert__() (datatypes.bits.Bit.Bit method), 4
__invert__() (datatypes.integers.Int8.Int8 method), 8
__invert__() (datatypes.integers.UInt8.UInt8 method), 12
__le__() (datatypes.integers.Int8.Int8 method), 8
__le__() (datatypes.integers.UInt8.UInt8 method), 12
__len__() (datatypes.integers.Int8.Int8 method), 8
__len__() (datatypes.integers.UInt.UInt method), 11
__len__() (datatypes.integers.UInt8.UInt8 method), 12
__lshift__() (datatypes.integers.Int8.Int8 method), 8
__lshift__() (datatypes.integers.UInt8.UInt8 method), 12
__lt__() (datatypes.integers.Int8.Int8 method), 9
__lt__() (datatypes.integers.UInt8.UInt8 method), 13
__mod__() (datatypes.integers.Int8.Int8 method), 9
__mod__() (datatypes.integers.UInt8.UInt8 method), 13
__mul__() (datatypes.bits.Bit.Bit method), 4
__mul__() (datatypes.integers.UInt8.UInt8 method), 13
__neg__() (datatypes.bits.Bit.Bit method), 4
__neg__() (datatypes.integers.Int8.Int8 method), 9
__neg__() (datatypes.integers.UInt8.UInt8 method), 13
__or__() (datatypes.bits.Bit.Bit method), 4
__or__() (datatypes.integers.Int8.Int8 method), 9
__or__() (datatypes.integers.UInt8.UInt8 method), 13
__repr__() (datatypes.bits.Bit.CryptoBit method), 5
__repr__() (datatypes.bits.Bit.PlainBit method), 6
__repr__() (datatypes.integers.Int8.Int8 method), 9
__repr__() (datatypes.integers.UInt8.UInt8 method), 13
__rshift__() (datatypes.integers.Int8.Int8 method), 9
__rshift__() (datatypes.integers.UInt8.UInt8 method), 13
__sub__() (datatypes.integers.Int8.Int8 method), 9
__sub__() (datatypes.integers.UInt.UInt method), 11
__sub__() (datatypes.integers.UInt8.UInt8 method), 14
__xor__() (datatypes.bits.Bit.Bit method), 4
__xor__() (datatypes.integers.Int8.Int8 method), 10
__xor__() (datatypes.integers.UInt8.UInt8 method), 14

A

AND() (datatypes.bits.Bit.Bit method), 3
AND() (datatypes.bits.Bit.CryptoBit method), 4
AND() (datatypes.bits.Bit.PlainBit method), 6

B

BadRightOperand, 7
Bit (class in datatypes.bits.Bit), 3
BitwiseOperationError, 3

C

completeAddOnOneBit() (in module datatypes.integers.Utility), 14
completeSubOnOneBit() (in module datatypes.integers.Utility), 15
CryptoBit (class in datatypes.bits.Bit), 4

D

datatypes.bits.Bit (module), 3
datatypes.bits.errors.BitwiseOperationError (module), 3
datatypes.bits.errors.InstantiationError (module), 3
datatypes.bits.errors.OpNotAllowedError (module), 3
datatypes.integers.error.BadRightOperand (module), 7
datatypes.integers.error.OverflowError (module), 7
datatypes.integers.Int8 (module), 7
datatypes.integers.UInt (module), 10
datatypes.integers.UInt8 (module), 11

datatypes.integers.Utility (module), 14
debug__printAsBoolean() (datatypes.bits.Bit.Bit method), 4
debug__printAsBoolean() (datatypes.bits.Bit.CryptoBit method), 5
debug__printAsBoolean() (datatypes.bits.Bit.PlainBit method), 6
debug_showValue() (datatypes.integers.Int8.Int8 method), 10
debug_showValue() (datatypes.integers.UInt.UInt method), 11
debug_showValue() (datatypes.integers.UInt8.UInt8 method), 14
decode() (in module demo.ImageUtils), 15
decrypt() (datatypes.bits.Bit.CryptoBit method), 5
decrypt() (datatypes.integers.Int8.Int8 method), 10
decrypt() (datatypes.integers.UInt8.UInt8 method), 14
decrypt() (in module demo.ImageUtils), 15
demo.ImageUtils (module), 15
dump() (in module demo.ImageUtils), 15

E

encode() (in module demo.ImageUtils), 15
encrypt() (datatypes.bits.Bit.PlainBit method), 7
encrypt() (datatypes.integers.Int8.Int8 method), 10
encrypt() (datatypes.integers.UInt8.UInt8 method), 14
encrypt() (in module demo.ImageUtils), 16

I

InstantiationError, 3
Int8 (class in datatypes.integers.Int8), 7

L

load() (in module demo.ImageUtils), 16

M

main (module), 16

N

negate() (in module demo.ImageUtils), 16
NOT() (datatypes.bits.Bit.Bit method), 3
NOT() (datatypes.bits.Bit.CryptoBit method), 5
NOT() (datatypes.bits.Bit.PlainBit method), 6

O

OpNotAllowedError, 3
OR() (datatypes.bits.Bit.Bit method), 4
OR() (datatypes.bits.Bit.CryptoBit method), 5
OR() (datatypes.bits.Bit.PlainBit method), 6
OverflowError, 7

P

PlainBit (class in datatypes.bits.Bit), 5

R

refresh() (datatypes.bits.Bit.CryptoBit method), 5
result (in module main), 16

S

setNoise() (datatypes.bits.Bit.CryptoBit method), 5
setUp() (testing.TestInt8Methods.TestInt8Methods method), 17
setUp() (testing.TestUInt8Methods.TestUInt8Methods method), 17
setUp() (testing.TestUIntMethods.TestUIntMethods method), 18
showValue() (datatypes.integers.UInt8.UInt8 method), 14

T

test_and() (testing.TestBitsMethods.TestBitsMethods method), 16
test_int8_abs() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_addition() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_debugshowValue() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_division() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_eq_ne() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_ge() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_gt() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_le() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_lshift() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_lt() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_mod() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_rshift() (testing.TestInt8Methods.TestInt8Methods method), 17
test_int8_subtraction() (testing.TestInt8Methods.TestInt8Methods method), 17
test_or() (testing.TestBitsMethods.TestBitsMethods method), 16
test_uint8_addition() (testing.TestUInt8Methods.TestUInt8Methods method), 17

test_uint8_debugshowValue() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_division() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_eq_ne() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_ge() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_gt() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_le() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_lshift() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_lt() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_mod() (testing.TestUInt8Methods.TestUInt8Methods method), 17
test_uint8_rshift() (testing.TestUInt8Methods.TestUInt8Methods method), 18
test_uint8_subtraction() (testing.TestUInt8Methods.TestUInt8Methods method), 18
test_uint_addition() (testing.TestUIntMethods.TestUIntMethods method), 18
test_uint_debugshowValue() (testing.TestUIntMethods.TestUIntMethods method), 18
test_uint_subtraction() (testing.TestUIntMethods.TestUIntMethods method), 18
test_xor() (testing.TestBitsMethods.TestBitsMethods method), 16
TestBitsMethods (class in testing.TestBitsMethods), 16
testing.TestBitsMethods (module), 16
testing.TestInt8Methods (module), 17
testing.TestUInt8Methods (module), 17
testing.TestUIntMethods (module), 18
TestInt8Methods (class in testing.TestInt8Methods), 17
TestUInt8Methods (class in testing.TestUInt8Methods), 17
TestUIntMethods (class in testing.TestUIntMethods), 18
toInt8() (datatypes.integers.UInt8.UInt8 method), 14
toUInt8() (datatypes.integers.Int8.Int8 method), 10

U

UInt (class in datatypes.integers.UInt), 10
UInt8 (class in datatypes.integers.UInt8), 11

X

XOR() (datatypes.bits.Bit.Bit method), 4
XOR() (datatypes.bits.Bit.CryptoBit method), 5
XOR() (datatypes.bits.Bit.PlainBit method), 6