

[Dashboard](#) / [My courses](#) / [COSC264](#) / [Week 1: Quiz \(Bit fiddling\) and Sheet \(Linux command line\)](#)
/ [Quiz: Number representations and bit-fiddling](#)

Started on Monday, 19 July 2021, 2:45 PM

State Finished

Completed on Thursday, 22 July 2021, 4:54 PM

Time taken 3 days 2 hours

Marks 15.90/17.00

Grade 9.35 out of 10.00 (94%)

Information

Later in the first term you will have to write Python3 programs for networking purposes, in particular the construction and inspection of so-called packet headers – this will be important both in the assignment and the superquiz for the first term. These operations require some understanding of how numbers are represented in memory and familiarity with "bit-twiddling" operations like bitwise AND, OR, XOR, shifting and other operations. If you are already familiar with these operations and know what the width of a number, and the one's complement or two's complement representation of signed integers is, then you will not need to do this quiz, perhaps with the exception of the last page. All others should.

This quiz is optional and is not marked.

Information

Computers have built-in capabilities for calculations with numbers of different kinds. Each number type requires a certain number of bits to store values of this type, this required number of bits is also often referred to as the **width** of a numerical type. Some of the most important types of numbers include:

- Signed integer numbers of a given width. A signed integer can take both positive and negative integer values. In today's computers signed integer types are usually of 32 bit or 64 bit width.
- Unsigned integer types of a given width. An unsigned integer can only take non-negative integer values, i.e. zero and positive values. Common types of unsigned integers include the byte (8 bit), and 32- or 64-bit unsigned integer values.
- Floating point numbers with 32 bits (single precision) or 64 bits (double precision)

We will only deal with signed and unsigned integer numbers in the following.

Information

An unsigned integer number is defined to be a non-negative integer, i.e. it can take the values 0, 1, 2, ..., up to some maximum number. If the width of our unsigned data type is w bits, then the maximum number is $2^w - 1$. For example, for bytes with $w=8$, the maximum number is 255, for $w=32$ the maximum number is 4,294,967,295. Note that due to some historical mishap, some older networking folks occasionally also speak about 'octets' when they mean a byte.

Question **1**

Correct

Mark 1.00 out of 1.00

What is the maximum number for a 24-bit wide unsigned integer?

Answer: 

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

What is the maximum number for a 16-bit wide unsigned integer?

Answer: 

Correct

Marks for this submission: 1.00/1.00.

Information

Suppose we are given some value x of an unsigned integer type with w bits width. We already learn as children how to write x as a decimal number, and we learn doing various arithmetic operations (e.g. addition, multiplication) in the decimal number system.

What does it mean when we say that a number x can be written in the decimal system as "1234"? This means that x can be written as the following sum:

$$x = 1 * 1000 + 2 * 100 + 3 * 10 + 4 * 1 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$$

So we write x as a sum of thousands (10^3), hundreds (10^2), tens (10^1) and ones (10^0). We say that the number 10 is the **base** for this representation of x , as we represent x in terms of powers of this base, and each of these powers can be multiplied with a factor between zero and nine – we call the latter also the **digits**.

A decimal number then is a base-10 number with the possible digits 0, 1, 2, ..., 9. When writing our number we normally only write as many digits as are needed to ensure that the leftmost digit is non-zero and leave away leading zeros. That means, we usually just write "1234" instead of "01234" or "001234".

The decimal or base-10 system is natural to humans because we have ten fingers (in a shoe-less society people might find base-20 numbers more convenient). Can we actually work with other bases? Or more precisely: how can we represent a given unsigned integer number x with respect to a given base b , with $b > 1$? Our task can be described as follows: find coefficients $a_0, a_1, a_2, \dots, a_m$ such that we can write

$$x = a_m * b^m + a_{m-1} * b^{m-1} + a_{m-2} * b^{m-2} + \dots + a_1 * b^1 + a_0 * b^0$$

We will see below that there is a procedure which can calculate these coefficients uniquely. This procedure only calculates as many coefficients a_i as needed, and the highest coefficient a_m will be different from zero. To show the number in base b to humans, we write the coefficients in the order $a_m a_{m-1} \dots a_1 a_0$ and each coefficient a_i is represented by a suitable digit. When defining how to write a single digit, we usually prefer methods in which each digit is represented by a single literal or character. For base 10 we use the literals '0' to '9', whereas for base 16 (the **hexadecimal number system**) we use the literals '0', '1', ..., '9', 'A', 'B', ..., 'F'.

Conversely, when we are given a sequence of literals like for example ' $x=1234$ ', then we need to know the basis b to actually determine the number x : in the decimal system, '1234' denotes the integer 1234, whereas in the hexadecimal '1234' actually denotes the integer 4660 (we have $4660 = 1 * 16^3 + 2 * 16^2 + 3 * 16^1 + 4 * 16^0$). Because of this ambiguity, many programming languages (including Python) allow to specify the base when the programmer writes a number literal. Commonly:

- '0x...' signifies that the literals given by '...' specify a hexadecimal number ($b=16$)
- '0b...' signifies that the literals given by '...' specify a binary number ($b=2$)
- '0o...' signifies that the literals given by '...' specify an octal number ($b=8$)

As a convention, when no prefix of the form '0x' or similar is given, then the base is $b=10$.

One important concept to take away from this is that a number as such is just a number and not intrinsically tied to any specific way to represent it or to any basis – the same number can be represented as a decimal, octal, hexadecimal or as a binary number, or perhaps in any other basis that someone finds useful (maybe Martians have seven fingers). We can in fact take two numbers written in two different representations and test them for equality. For example, you can use your Python interpreter to check that '0x20 == 32' is true and '0x20 == 20' is false. With this, a sentence like ' x is a hexadecimal number' is, strictly speaking, nonsensical and should be replaced by a statement like 'the number x is represented as a hexadecimal number'.

In the context of programming languages there is a further layer which may lead to confusion. When in Python you enter the text '0x20 == 32' into the command line and press ENTER, then in the first stage the interpreter will interpret this test simply as a string (i.e. a sequence of characters), and not yet as a number. Only during the further processing will the string '0x20' be converted into a number, and this conversion will be guided by the leading '0x' to tell the interpreter that the given string signifies an integer number in hexadecimal representation.

Information

With all the previous explanations in place, it is now simple to describe how unsigned integer numbers are represented in the computer memory: as a base-2 number of a certain width w (e.g. 32 bit or 64 bit), with the digits 0 and 1. One such digit is called a **bit**. At the lowest level, all processing of integer numbers within a processor (e.g. addition, multiplication) works with bits.

Question 3

Correct

Mark 1.00 out of 1.00

Write a Python3 function which takes two parameters: an unsigned integer x (for Python: an integer $x \geq 0$) and a positive integer *base*. This function should behave as follows:

- First check the type of parameter x : if it is not an integer, return -1
- Next check the type of parameter *base*: if it is not an integer, return -2
- Third, check if x is ≥ 0 . If not, then return -3
- Fourth, check if *base* is ≥ 2 . If not, then return -4
- Otherwise, find the coefficients for the number x in the given base, as integer values. Find only as many coefficients as are needed and collect them in a result list, with the highest-order coefficient being the first element, the second-highest-order coefficient being the second element, and so forth. Return the list of coefficients.

In writing your function you should use the Python operators `//` (integer division) and `%` (modulo operator).

For example:

Test	Result
<code>print (convert(1234, 10))</code>	<code>[1, 2, 3, 4]</code>
<code>print (convert(4660, 16))</code>	<code>[1, 2, 3, 4]</code>

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```

1 def convert (x, base):
2
3     if type(x) != int:
4         return -1
5
6     if type(base) != int:
7         return -2
8
9     if not x >= 0:
10        return -3
11
12    if not base >= 2:
13        return -4
14
15    else:
16        return find_coefficients(x, base)
17
18 def find_coefficients(x, base):
19
20     n = len(str(x)) - 1
21     result = []
22     while x > 0:
23         remainder = x % base
24         result.append(remainder)
25         x = x // base
26     result.reverse()
27     return result

```

	Test	Expected	Got	
✓	<code>print (convert(1234, 10))</code>	<code>[1, 2, 3, 4]</code>	<code>[1, 2, 3, 4]</code>	✓
✓	<code>print (convert(4660, 16))</code>	<code>[1, 2, 3, 4]</code>	<code>[1, 2, 3, 4]</code>	✓
✓	<code>print (convert(1234, 9))</code>	<code>[1, 6, 2, 1]</code>	<code>[1, 6, 2, 1]</code>	✓
✓	<code>print (convert(1234, 2))</code>	<code>[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0]</code>	<code>[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0]</code>	✓
✓	<code>print (convert(8588, 10))</code>	<code>[8, 5, 8, 8]</code>	<code>[8, 5, 8, 8]</code>	✓



	Test	Expected	Got	
✓	print (convert(8588, 7))	[3, 4, 0, 1, 6]	[3, 4, 0, 1, 6]	✓
✓	print (convert(8588, 2))	[1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0]	[1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0]	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question 4

Correct

Mark 1.00 out of 1.00

Use the previous function 'convert' to write a Python function to convert an unsigned integer x (in Python: a non-negative integer) into a string representing the hexadecimal representation of x . This function

- Should first check if the argument x is an integer. If not, return -1
- Should next check if the argument x is non-negative. If not, return -2
- Otherwise, convert the argument into a hexadecimal string, using the 'convert' function. The string should be prepended with '0x'. Use 'A', 'B', ..., 'F' for the literals greater than or equal to ten.

For example:

Test	Result
<code>print(hexstring(1234))</code>	0x4D2

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```

1 def hexstring(x):
2     if type(x) != int:
3         return -1
4
5     if not x >= 0:
6         return -2
7
8     else:
9         return find_hex(x)
10
11 def find_hex(x):
12     hex_list = convert(x, 16)
13
14     mapping = ['A', 'B', 'C', 'D', 'E', 'F']
15     result = '0x'
16     for i in hex_list:
17
18         if i > 9:
19             result += mapping[i-10]
20         else:
21             result += str(i)
22
23     return result
24
25 def convert(x, base):
26
27     if type(x) != int:
28         return -1
29
30     if type(base) != int:
31         return -2
32
33     if not x >= 0:
34         return -3
35
36     if not base >= 2:
37         return -4
38
39     else:
40         return find_coefficients(x, base)
41
42 def find_coefficients(x, base):
43
44     n = len(str(x)) - 1
45     result = []
46     while x > 0:
47         remainder = x % base
48         result.append(remainder)
49         x = x // base
50     result.reverse()
51     return result

```



	Test	Expected	Got	
✓	print(hexstring(1234))	0x4D2	0x4D2	✓
✓	print(hexstring(12345))	0x3039	0x3039	✓
✓	print(hexstring(123456))	0x1E240	0x1E240	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Unsigned integers can only represent non-negative integer numbers, i.e. the numbers 0, 1, 2, ... and so on, up to a maximum number. To allow for negative integer numbers as well, we will have to use **signed integers**, which have a different representation.

Practically all processors today natively support calculations with unsigned integers. Recall that within a computer an unsigned integer number is represented as a block of bits, e.g. of 32 bits or 64 bits width. Since this is all a computer really can do, we will have to represent negative integers also with just using a sequence of bits, there are no "extra hardware markers" or some such to record the sign of an integer value.

The most common method to represent a negative integer number (while playing by the rules for unsigned integers) is the so-called **two's complement** method, best explained by an example: suppose we are given the (positive) integer number 37 and store this as an eight bit number '00100101'. To convert this into a representation of the integer number -37, we proceed as follows:

1. **Invert all the bits:** inverting '00100101' bit-by-bit gives '11011010'
2. **Add one to the result:** perform a standard unsigned integer addition operation to add one. In our example, adding '00000001' to '11011010' gives the result '11011011'. It should be mentioned that this addition is constrained to eight bits without carry.

And that's it! The result obtained this way is the binary representation of the number -37. Note that in this method of representation any negative integer number has the leftmost bit set to one, which makes this bit effectively the sign bit for the number. The beauty of this representation is that one can apply the standard methods for addition, subtraction, multiplication and division in the same way as one would for unsigned integers, and one gets correct results.

A few examples for numbers in two's complement representation (using eight bits):

- The integer 0 is represented as '00000000'
- The integer 1 is represented as '00000001'
- The integer -1 is represented as '11111111' ($255 = 256 - 1$ in unsigned interpretation, see if you can confirm this!)
- The integer 6 is represented as '00000110'
- The integer -6 is represented as '11111010' ($250 = 256 - 6$ in unsigned interpretation, see if you can confirm this!)
- The integer -128 is represented as '10000000'
- The integer 127 is represented as '01111111'

In general, the two's complement representation of a strictly negative number x using n bits can actually be understood as the unsigned integer representation of the number $2^n - x$ when using just n bits. As a more complete example, the following table shows all possible values in two's complement that a four-bit number can take:

Signed two's-complement bit pattern Corresponding decimal value

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8 ($=2^4-8$)
1001	-7 ($=2^4-7$)
1010	-6 ($=2^4-6$)
1011	-5 ($=2^4-5$)
1100	-4 ($=2^4-4$)
1101	-3 ($=2^4-3$)
1110	-2 ($=2^4-2$)
1111	-1 ($=2^4-1$)

Signed two's complement values for four-bit numbers

As you can see from this example, an n -bit number in two's-complement representation can hold 2^{n-1} strictly negative values (which all have the highest bit set to one), $2^{n-1}-1$ strictly positive values, and zero.

Question **5**

Correct

Mark 1.00 out of 1.00

Give the two's complement representation of -56 as an eight bit number. Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 11001000



Correct

Marks for this submission: 1.00/1.00.


Question **6**

Correct

Mark 1.00 out of 1.00

You are given the following binary representation of an eight-bit number: '10110010' in two's complement. Determine whether the following claim is true or false: The number is positive.

Select one:

- ☐ True
- ☒ False 

Recall that in two's complement a negative number always has a 1 in the leftmost bit.

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 1.00 out of 1.00

You are given the following eight-bit number in two's complement: '00101100'. Give the decimal value of this number.

Answer: 44



Correct

Marks for this submission: 1.00/1.00.

Question **8**

Correct

Mark 0.33 out of 1.00

You are given the following eight-bit number in two's complement: '10111101'. Give the decimal value of this number.

Answer: 

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.33/1.00**.Question **9**

Correct

Mark 0.67 out of 1.00

Suppose we have a weird computer which uses integer numbers of 19 bits width. What is the largest positive integer number that can be represented? Please give your answer as a decimal number.

Answer: 

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.67/1.00**.Question **10**

Correct

Mark 1.00 out of 1.00

Assuming the same weird 19-bit computer: What is the largest negative integer number that can be represented? Please give your answer as a decimal number.

Answer: 

Correct

Marks for this submission: 1.00/1.00.

We assume that you are vaguely familiar with propositional logic. In propositional logic, one deals with logical statements and logical operators between these. A logical statement, e.g. P , is a statement to which we can uniquely assign one of the two truth values 'False' or 'True'. As an example, the statement ' $2+2=5$ ' is a logical statement, since we can objectively say whether it is true or false (once we have clarified a suitable context for this statement, e.g. that we are operating in the "normal" integer number space). In contrast, the statement "This rose smells beautiful" cannot be objectively answered and hence is not a logical statement, neither is "What day is it today?"

Logical statements can be combined with logical operators, the most well-known of these are AND, OR, NOT, XOR. The three operators AND, OR and XOR take two logical statements as "input" and calculate from their individual truth values the truth value of the combined statement. These operators are usually represented as tables. The table for the logical AND, OR, XOR and NOT is:

P	Q	P AND Q	P OR Q	P XOR Q	NOT P
True	True	True	True	False	False
True	False	False	True	True	False
False	True	False	True	True	True
False	False	False	False	False	True

Logical Operations

We now make a tiny but decisive step: instead of representing the truth values by 'True' and 'False', we represent them by the numbers 0 and 1: we assign the value 1 to 'True' and the value 0 to 'False'. We can now represent the truth value for a logical statement by a single bit, and the above four logical operations can be represented as:

P	Q	P AND Q	P OR Q	P XOR Q	NOT P
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

Logical operations as bits

Hence, they become operations on bits. Modern processors and most modern programming languages support these logical operations on bits, but usually take this even a step further by not performing these calculations on individual bits, but rather on blocks of bits, e.g. blocks of eight, 16, 32 or 64 bits. As an example, the **bitwise-AND** operation on eight-bit numbers takes two eight-bit values x and y as input and produces an eight-bit output in which the processor has performed a 'logical AND' in a bit-by-bit fashion: the leftmost bit in the result is the logical AND between the leftmost bit of x and the leftmost bit of y , the second-leftmost bit in the result is the logical AND between the second-leftmost bit of x and the second-leftmost bit of y , and so on.

As an aside, note that the result of the AND operation on two bits actually agrees with the result of a multiplication carried out on the input values: 0 times 0 gives 0, 0 times 1 and 1 times 0 give 0, and 1 times 1 gives 1. Convince yourself about this.

Python 3 supports the following bitwise logical operators:

- ' $x \& y$ ' is the bitwise AND.
- ' $x | y$ ' is the bitwise OR.
- ' $x \wedge y$ ' is the bitwise XOR.
- ' $\sim x$ ' is the bitwise NOT.

And while we are at it, we introduce two further bitwise operators that Python supports:

- ' $x \ll n$ ' is the bitwise shift to the left: suppose we are given an eight-bit binary value $x = b_7b_6b_5b_4b_3b_2b_1b_0$ and an integer n between 0 and 7. The result of this operation is an eight-bit binary value y in which the bits of x are shifted to the left by n bits such that on the right we fill in zero bits and on the left we lose the bits that are "shifted out" of the allocated eight bit (in the Python programming language this is not entirely accurate, as Python would simply add more bits, but still the bit b_7 would not appear in the lowest eight bits anymore after the shifting operation). So, for example ' $x \ll 3$ ' becomes $y = b_4b_3b_2b_1b_0000$
- ' $x \gg n$ ' is the bitwise shift to the right: this is very similar to ' $x \ll n$ ' but zeros are filled in from the left and bits are "shifted out" on the right. For example, ' $x \gg 3$ ' becomes $y = 000b_7b_6b_5b_4b_3$

Question **11**

Correct

Mark 1.00 out of 1.00

Calculate the bitwise AND between the two eight-bit numbers '01011101' and '11001001'. Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 01001001



Correct

Marks for this submission: 1.00/1.00.

Question **12**

Correct

Mark 1.00 out of 1.00

Calculate the bitwise OR between the two eight-bit numbers '01011101' and '11001001'. Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 11011101



Correct

Marks for this submission: 1.00/1.00.

Question **13**

Correct

Mark 1.00 out of 1.00

Calculate the bitwise XOR between the two eight-bit numbers '01011101' and '11001001'. Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 10010100



Correct

Marks for this submission: 1.00/1.00.

Question **14**

Correct

Mark 1.00 out of 1.00

Calculate the left-shift by three places of the eight-bit number '01011101' . Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 11101000



Correct

Marks for this submission: 1.00/1.00.

Question **15**

Correct

Mark 1.00 out of 1.00

Calculate the right-shift by three places of the eight-bit number '01011101' . Enter the result as a sequence of eight zeros and ones without anything else (e.g. '10101010' without the quotes).

Answer: 00001011



Correct

Marks for this submission: 1.00/1.00.

Why do we bother with bitwise-AND, left-shifts and the like?

COSC 264 is a course about networking. A key activity in networking involves the transmission of 'messages' between two communication partners over a network. Such a 'message' – or **packet** in networking lingo – is a sequence of bits and/or bytes that has a certain length. What we say here about packets is not the whole truth, but simplified. One part of such a packet, the **data part**, contains actual data that is of interest to a user (e.g. parts of a cat video), and other parts of such a packet are for control purposes, for example to help the network with finding the right way for a packet or checking its data integrity. We call these latter parts the **control part**. The control part is made up of a number of different pieces of information, which we also refer to as **fields**. As an example, consider a postcard you send to a relative:

- In the data part the postcard contains some text describing how bad the weather or how poor the food is at your location.
- In the control part the postcard contains addressing information describing the address of the recipient, and optionally also the address/name of the sender. Additionally, there is a stamp. The recipient address consists of a number of fields, including: the recipient's name, street name, house number, city, zip code, country. The control part (in particular the recipient address) is used by the postal service to deliver the postcard to its final destination.

In a networking packet, often a fixed number of bits is allocated to the control part, and in fact the designers of networking software often try to use only the minimum amount of bits required for a given purpose. Each field of the control part occupies some of these bits at prescribed (usually contiguous) locations, distinct from the bits allocated to other fields. As a simple example we could allocate a total of 424 bits to the recipient address field on a postcard as follows:

- The first 128 bits are reserved for the recipient name
- The next 128 bits are reserved for the street name
- The next 16 bits are reserved for the house number
- The next 128 bits are reserved for the city
- The next 16 bits are reserved for the zip code
- The final 8 bits are reserved for the country (code).

In a packet, these 424 bits would be stored as one single contiguous block of bits, sub-divided into the six fields as above.

The transmission and reception of packets is governed by so-called **protocols**. A protocol specifies, amongst other things, which types of packets are defined, which fields a particular type of packet contains and their order, how many bits these fields occupy, which values are allowed in a field and how the values for a field are encoded, how transmitted packets must be constructed and how received packets need to be processed.

When putting together a packet for transmission, the sending station must fill in sensible values to the different fields making up the control and data parts of the packet.

When processing a received packet, the receiving station needs to extract the user data and the fields of the control part and decide its next action based on that. Before performing any action though, the packet is usually checked for correctness, e.g. whether all the fields have an allowed value in them. If a packet is found incorrect, then usually it is discarded.

In both constructing packets and in extracting data from packets the bitwise operations that we have discussed so far come in very handy. In the last part of this quiz we will consider a few typical tasks in packet construction / deconstruction and how these can be solved using the operations we have studied so far.

Question 16

Correct

Mark 0.90 out of 1.00

In our first example we consider a networking system in which we transmit dates. To save space, a date is represented as a 32-bit number as follows:

- The first four bits (or: the four highest-valued bits in the 32-bit number) represent the month. The four-bit value 0 stands for January, 1 stands for February and so forth.
- The next five bits represent the day. The five-bit value 0 stands for the first day of a month, 1 stands for the second day of a month, and so on.
- The remaining 23 bits represent the year, counting from year 0 onwards. So, the year 2010 would be stored as the 23-bit number 2010.

We will be given a 32-bit number x . How do we extract the values of the three fields for month, day and year out of that number? Let us look at one example, the month field. To extract the (value of the) month field, we proceed in two steps:

- We first 'mask out' from x all the bits that do not belong to the month field by setting them to zero. The way to achieve this is to calculate the bitwise-AND between x and the "bitmask" 11110000 00000000 00000000 00000000 given here in binary representation with extra spaces for clarity (or in hex 0xF0000000). In this bitmask all bits but the initial four are zero. When we perform this operation and call the result y , then the highest four bits of y are identical to the highest four bits of x , and the remaining 28 bits of y are zero (check for yourself!).
- In the number y we obtained at the end of the first step, only the highest four bits have a chance of being different from zero – and if the highest four bits would be, for example, 0101, then the value of y as an integer value would be 1342177280, certainly not a number in the allowed range between 0 and 11 for a month. To achieve our goal we shift the value of y to the right by $32-4=28$ bits.

In summary, the value of the month field becomes (in Python3 syntax) `(x & 0xF0000000) >> 28`. But we still need to make one further step: remember that we actually have chosen to encode the months as numbers from 0 to 11, which is not "natural" to us. Rather, we are used to number the months from 1 to 12. So, we need to add a '1' to the result of our computation.

With this background, write a Python3 function which takes a 32-bit number x and extracts the month, day and year fields, and which returns the given date as a string in the following format: 'dd.mm.yyyy', i.e. as numbers separated by dots. In the case of month and day you should leave away any leading zeros. Some example outputs would be '5.5.2017' for the 5th of May 2017, or '19.12.3266' for the 19th of December, 3266.

For example:

Test	Result
<code>print(decodedate(1107298273))</code>	5.5.2017

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```

1 def decodedate (x):
2
3     month_bin = x & 0xF0000000
4     month = (month_bin >> 28) + 1
5
6     day_bin = x & 0x0F800000
7     day = (day_bin >> 23) + 1
8
9     year = (x & 0x007FFFFF)
10
11     return str(day) + '.' + str(month) + '.' + str(year)
12
13
14 def hexstring (x):
15     if type(x) != int:
16         return -1
17
18     if not x >= 0:
19         return -2
20
21     else:
22         return find_hex(x)
23
24 def find_hex(x):
25     hex_list = convert (x, 16)
26
27     mapping = ['A', 'B', 'C', 'D', 'E', 'F']
28     result = '0x'
29     for i in hex_list:
30         result += mapping[i]
31 
```



```

31 |         if i > 9:
32 |             result += mapping[i-10]
33 |         else:
34 |             result += str(i)
35 |
36 |     return result
37 |
38 | def convert(x, base):
39 |     """ converts decimal -> binary """
40 |     if type(x) != int:
41 |         return -1
42 |
43 |     if type(base) != int:
44 |         return -2
45 |
46 |     if not x >= 0:
47 |         return -3
48 |
49 |     if not base >= 2:
50 |         return -4
51 |
52 |     else:
53 |         return find_coefficients(x, base)
54 |
55 | def find_coefficients(x, base):
56 |
57 |     n = len(str(x)) - 1
58 |     result = []
59 |     while x > 0:
60 |         remainder = x % base
61 |         result.append(remainder)
62 |         x = x // base
63 |     result.reverse()
64 |     return result
65 |

```

	Test	Expected	Got	
✓	print(decodeddate(1107298273))	5.5.2017	5.5.2017	✓
✓	print(decodeddate(2298488591))	19.9.9999	19.9.9999	✓
✓	print(decodeddate(998246312))	24.4.1960	24.4.1960	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.90/1.00**.

Question 17

Correct

Mark 1.00 out of 1.00

Now we consider the converse problem: putting together a 32-bit encoded date value from three given values for the day, month and year. Again, we will put bitwise operations to good use. Let us consider the month field in some more detail. Suppose we are given a value m between 0 and 11 representing a month value. The bit representation of m is guaranteed to have at most the rightmost four bits different from zero. Suppose furthermore we have a 32-bit number x , in which we want to set the four highest-valued bits to the same value as the four lowest-valued bits of m , while leaving the remaining bits of x untouched (for example because they already contain the right bit patterns for the day and the year). We achieve this goal in three steps:

- We first shift the value of m to the left by 28 bits, so that now the contents of bits 1 to 4 (the leftmost four bits) move into bits 29 to 32 (the rightmost four bits in a 32-bit number). Call the resulting number m^* and remember that the rightmost 28 bits of m^* are zero, since the left-shift operation fills in zero-bits on the right.
- Next we "clean out" the highest four bits of x and set them to zero, to prepare for the third step. We achieve this by "masking out" the four highest bits by performing a bitwise-AND between x and $0x0FFFFFFF$. We call the result x^* .
- The final step is to "merge together" x^* and m^* by calculating their bitwise-OR.

Convince yourself carefully that these computation steps do indeed what we want. In Python3, the entire computation could be achieved with the following statement, in which for brevity we do not make the intermediate variables x^* and m^* explicit: `'x = (x & 0x0FFFFFFF) | (m << 28)'`. This computation is valid for the month being a value between 0 and 11 (including). If we want to do the same with month values between 1 and 12, we first have to translate them to the range 0 to 11, and the computation becomes `'x = (x & 0x0FFFFFFF) | ((m-1) << 28)'`.

With this background information, write a Python3 function which takes three values as parameters (one for day, one for month, one for year), checks them for the right ranges, and produces a 32-bit encoded value following the same specifications as in the previous question:

- The first four bits encode the month, with allowable month values from 0 to 11.
- The next five bits encode the day of the month, with allowable day values from 0 to 30.
- The final 23 bits encode the year, with allowable year values being non-negative and limited to $2^{23}-1$

This function should accept "natural" values for month and day of the month, i.e. values between 1 and 12 for the month and values between 1 and 31 for the day of the month. Your function should check the given parameter values for being within reasonable ranges (e.g. the 'day' argument needs to be an integer between 1 and 12, including), but you do not need to check the overall sanity of dates (e.g. your function can allow for a date like February 30). The return value is either -1 if any of the preconditions is violated (e.g. the day value given as parameter is 73), or it is an encoded 32-bit integer.

For example:

Test	Result
<code>print(encodeddate(5,5,2017))</code>	1107298273

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```

1 def encodeddate (day, month, year):
2
3     if day > 31 or month > 12 or day <= 0 or month <= 0:
4         return -1
5
6     d = (day-1) << 23
7     m = (month-1) << 28
8
9     md = d | m
10
11    mdy = md | year
12
13    return mdy
14
15 def decodedate (x):
16
17    month_bin = x & 0xF0000000
18    month = (month_bin >> 28) + 1
19
20    day_bin = x & 0x0F800000
21    day = (day_bin >> 23) + 1
22
23    year = (x & 0x007FFFFF)
24
25    return str(day) + '.' + str(month) + '.' + str(year)
26

```



```

27
28 def hexstring (x):
29     if type(x) != int:
30         return -1
31
32     if not x >= 0:
33         return -2
34
35     else:
36         return find_hex(x)
37
38 def find_hex(x):
39     hex_list = convert (x, 16)
40
41     mapping = ['A', 'B', 'C', 'D', 'E', 'F']
42     result = '0x'
43     for i in hex_list:
44
45         if i > 9:
46             result += mapping[i-10]
47         else:
48             result += str(i)
49
50     return result
51
52 def convert (x, base):
53     """ converts decimal -> binary """
54     if type(x) != int:
55         return -1
56
57     if type(base) != int:
58         return -2
59
60     if not x >= 0:
61         return -3
62
63     if not base >= 2:
64         return -4
65
66     else:
67         return find_coefficients(x, base)
68
69 def find_coefficients(x, base):
70
71     n = len(str(x)) - 1
72     result = []
73     while x > 0:
74         remainder = x % base
75         result.append(remainder)
76         x = x // base
77     result.reverse()
78     return result

```

	Test	Expected	Got	
✓	print(encodeddate(5,5,2017))	1107298273	1107298273	✓
✓	print(encodeddate(9,11,4444))	2751467868	2751467868	✓
✓	print(encodeddate(30,12,345752))	3196405400	3196405400	✓
✓	print(encodeddate(32,5,2017))	-1	-1	✓
✓	print(encodeddate(5,15,2017))	-1	-1	✓
✓	print(encodeddate(0,5,2017))	-1	-1	✓
✓	print(encodeddate(5,0,2017))	-1	-1	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

◀ Announcements

Jump to...





[Quiz: Networked applications, QoS](#) ►

