CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework: Hexadecimal Math

Hexadecimal Math

Modify your prior solution to the binary math problem, or start with the provided solution, so that the output is in hexadecimal instead of binary. It is *suggested* to use *lower case* letters in your hexadecimal (they are easier to distinguish from digits). Place your code in a file called `addsub.asm` and make sure to correct the first line to correctly compile your code!

```
$ ./addsub 9000000000000000000 8999999999999999999
Adding:
7ce66c50e2840000
7ce66c50e283ffff
f9ccd8a1c507ffff
Subtracting:
7ce66c50e2840000
7ce66c50e283ffff
0000000000000001
```

Uses a look-up table

The offset from nyb gives the correct hex digit.

```
                section .data
nyb             db "0123456789abcdef"
```

Not much else changed

We divide by different quantities, and just grab a single character from `[nyb + offset]`.

```
write_hex_qword:
        ; ...
        ; Get high nybble and divide by
sixteen.
        and rax, 0xf0
        shr rax, 4
        mov rdi, 1
        lea rsi, [nyb + rax]
        mov rdx, 1
        mov rax, 1
        syscall
        ; Restore the byte value.
        pop rax
        ; Get low nybble.
        and rax, 0xf
        mov rdi, 1
        lea rsi, [nyb + rax]
        mov rdx, 1
        mov rax, 1
        syscall
        ; ...
```

Python 3

Python 3

There is no reason to worry about Python 2; you can almost ignore it, except…

Some scripts will assume that `python` refers to Python 3, and some will assume that `python` refers to Python 2.

On Ubuntu, use `python3`.

You can safely remove Python 2 if you want.

```
# Remove python2
sudo apt remove python2
sudo apt autoremove

# Make python point to python3
sudo update-alternatives --install \
    /usr/bin/python python /usr/bin/python3.7 1

# Maybe install IPython
sudo apt install ipython3
```

Python

Python is an interpreted *and* compiled language. The Python compiler produces bytecode, much like Java.

```
prog.py → prog.pyc
```

Play around with it at the prompt, or online: https://repl.it/languages/python3

```
#!/usr/bin/env python3
#

# Comments and stuff go here…
# Define functions, classes, etc.

if __name__ == "__main__":
  # Stuff to do when this is run from the
  # prompt goes here.  Note that Python uses
  # indentation, not braces, to indicate
  # nested blocks.
```

Python

Python can be *dynamically* typed, but it is getting static typing as an option.

You can specify type annotations, but the interpreter will *not* check them (yet).

```
$ python3
>>> a = 5
>>> print(a)
5
>>> a = "five"
>>> print(a)
five
>>> type(a)
<class 'str'>
>>> a:int = "five"
>>> type(a)
<class 'str'>
```

Functions

Define functions with `def`.

Note that a colon (`:`) introduces the body of a function or a statement, and indentation is significant.

Return a value with `return`, if any.

Note the array slicing notation.

```
#!/usr/bin/env python3
#

import sys

def hello(name):
    print(f"Hello {name}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Missing name.")
    else:
        for name in sys.argv[1:]:
            hello(name)
```

Documentation

Strings can be written `"..."` or `'...'`.

You can also write multi-line strings using `"""` or `'''`.

Document a file, class, or function by including a string right after the declaration.

```
#!/usr/bin/env python3
#

'''Demonstrate some basic python.'''

import sys

def hello(name):
    '''Say hello to the given name.'''
    print(f"Hello {name}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Missing name.")
    else:
        for name in sys.argv[1:]:
            hello(name)
```

Documentation

You can get help using `help`.

A file is a module, and you can import it with `import`, assuming it is on the path or in the current directory.

You can list the content of a module with `dir`.

```
>>> import hello
>>> hello("Fred")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'module' object is not callable
>>> dir(hello)
['__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__',
'__package__', '__spec__', 'hello', 'sys']
>>> print(hello.__doc__)
Demonstrate some basic python.
```

Documentation

Using `dir` and `help` can tell you a lot about a package.

You should always write a documentation string for every file, class, and function you write.

```
>>> help(hello.hello)

Help on function hello in module hello:

hello(name)
    Say hello to the given name.
```

Basics

Floating-point and integer division use different operators, and do *not* depend on the types of the operands.

It is very easy to convert from strings to numbers and back.

```
>>> 5 / 2
2.5
>>> 5 // 2
2
>>> int(5/2)
2
>>> float(5//2)
2.0
>>> str(5/2)
'2.5'
>>> int('5')/float('2')
2.5
>>> hex(10)
'0xa'
>>> bin(10)
'0b1010'
>>> int(hex(0xff + 1), base=16)
256
>>> int(hex(0xff + 1), base=0)
256
```

Basics

Arithmetic operations are what you'd expect, and the order is what you'd expect, with a few interesting additions.

```
>>> 3^3    # xor
0
>>> 3^0
3
>>> 3%2    # remainder
1
>>> 5**2   # exponentiation
25
>>> 5.0**2
25.0
>>> 25**0.5
5.0
>>> eval('6*5')
30
```

If

If you want an empty block, you must use `pass`, which is a no-op.

```
if a >= 21:
    # Don't do anything.
    pass

elif a >= 10:
    print("At least 10")

elif a >= 5:
    print("At least 5")

else:
    print("Value too low")
```

Loops

You can have an `else` clause to the loop.

```
>>> a = 13
>>> while a>10:
...     print(a)
...     a -= 1
...
13
12
11

>>> for x in range(1,10,2):
...     print(x)
...
1
3
5
7
9
```

Exceptions

Use `raise` to raise an exception.

Use a `try` block to run code that might raise an exception, and `except` blocks to catch any exceptions.

Any `else` block is executed if there are no exceptions, and a `finally` block is always executed.

```
try:
    file = open('output.txt', 'w')
    file.write('Hello file')
except PermissionError as err:
    print(f"Bad permissions: {err}")
except:
    print("A bad thing happened")
else:
    print("Success")
finally:
    file.close()
```

```
with open('output.txt', 'w') as file:
    file.write('Hello file')
```

Classes

Classes can specify an optional base class in parentheses.

Instance methods take `self` as the first argument. Class methods do not.

Instance data is stored in `self`.

```
class Greeter(object):
    """Issue greetings."""

    def __init__(self, greeting):
        """Initialize an instance."""
        self.greeting = greeting

    def get_greeting(self, name):
        """Greet someone."""
        return self.greeting + " " + name

    def greet(self, name):
        """Greet on standard output."""
        print(self.get_greeting(name))

>>> g = Greeter("Bonjour")
>>> g.greet("Stan")
Bonjour Stan
```

Capstone & pyelftools

Install stuff

```
$ sudo apt install python3-pip
$ pip3 install capstone
$ pip3 install pyelftools
```

Test it out

```
$ python3
>>> import elftools
>>> import capstone
```

Homework:
Due: Tuesday, February 25

Entry Point

Modify the provided code to begin disassembly at the program's entry point. Call your program `entry_point.py`.

Try your code on some real programs and see how it does.
`./entry_point.py `which python3``

Give some thought to how you would handle the anti-disassembly code shown earlier.

Next time:
Computing Control Flow