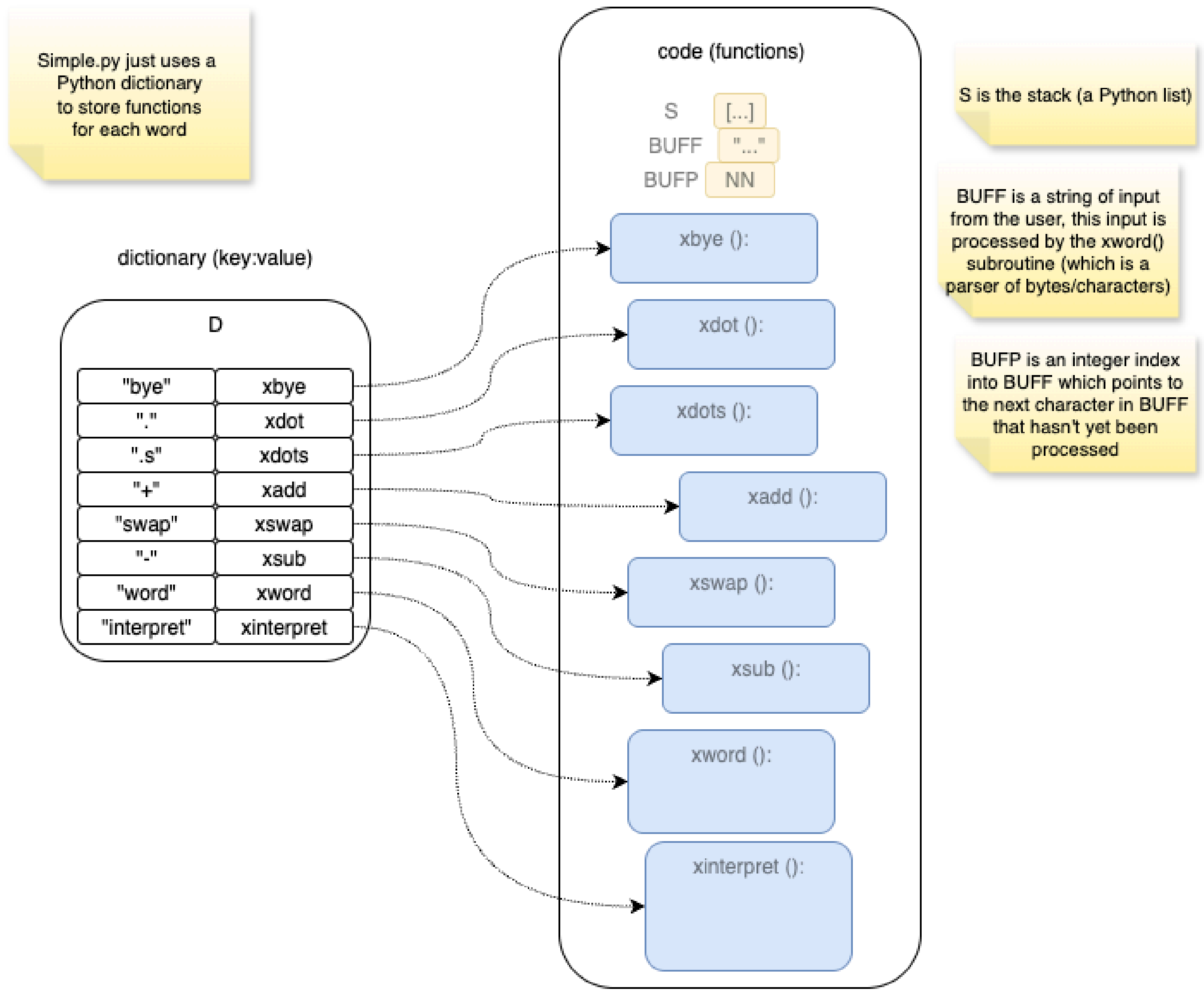
Simple.py just uses a Python dictionary to store functions for each word
dictionary (key:value)
D
"bye" xbye
"." xdot
".s" xdots
"+" xadd
"swap" xswap
"-" xsub
"word" xword
"interpret" xinterpret
code (functions)
S [...]
BUFF "..."
BUFP NN
xbye ():
xdot ():
xdots ():
xadd ():
xswap ():
xsub ():
xword ():
xinterpret ():
S is the stack (a Python list)
BUFF is a string of input from the user, this input is processed by the xword() subroutine (which is a parser of bytes/characters)
BUFP is an integer index into BUFF which points to the next character in BUFF that hasn't yet been processed

Usage

To run the `simple.py` REPL (Read Eval Print Loop) at the command line, type:

```
python3 simple.py
```

Code Walkthrough

The following is a copy of `simple.py` with comments inserted after every significant section.

Header

~~~
```python
APP = dict(
~~~

```
    name="Simple Forth-like",
    about="Simple Forth-style parser and evaluator.",
    review=(("This is a beautifully minimal implementation of a Forth-like
stack-based interpreter in Python!", "Claude Code"),))
```

Header describing the project.

Stored as a dictionary in the runtime code with 3 keys, "name", "about", and "review".

The code can look at the value of the variable `APP`. This version of the `simple.py` does not use this variable, but, if we changed the last few lines to

```
if "__main__" == __name__:
    print (APP)
    ok()
```

it would print the contents of the header before going into the interpreter REPL (Read Eval Print Loop)

The Forthish Stack

```
S = [];
```

`S` is the Forthish stack. Here, we initialize it to the empty list.

In this simple Python version, we use a list as the stack. The "top" of the stack is the last element of the list. This is Python convention.

In practice, when using just a basic CPU, we would treat memory as an array of bytes. In this case, we are free to use whatever convention we wish. For example we might allocate an array of 4 kilobytes (4,096 bytes) with indices of 4,096-8,191 (indices into the memory array are called "pointers"), and, we might imagine that the bottom of the stack is at 8,192 and grows "upwards" towards 4,096.

On a 16-bit CPU, we need to use 16 bit integers as indices (pointers). Hence, an index is 2 bytes (bytes are 8 bits each, 16 = 8 + 8). It is convenient to make *everything* on the stack be 16 bits, by convention. Hence, an element on the stack can be interpreted as a 16-bit index (pointer), or a 16-bit integer (0-65,535), or a 15-bit signed integer (-65535 → 0 → 65535), or a 16-bit index into a string pool.

Input Buffer

```
BUFF = "";
```

Declaration of the `BUFF` input string buffer. Initialized to be the empty string.

```
BUFP = 0
```

Index of the next unread character in `BUFF`.

Initialized to 0.

`P` is used as a suffix because, historically, indices into RAM are called *pointers*.

`BUFF` and `BUFP` are manipulated by the `xword` subroutine. When the buffer is empty or if BUFP points past the end of `BUFF`, more input is read in from the user at the command line and `BUFP` is reset to 0. The input is stored in `BUFF`.

Word Implementations as Python Subroutines

In Python, a *docstring* can be supplied as the first item in a function body (a string).

In Forth, it is a convention to show the before and after contents of the stack inside parentheses, separated by `--`.

The stack picture notation shows arguments in the order they're added to the stack -- so (assuming stack that grows rightwards (TOS is Top of Stack)):

```
OK 3
( 3 TOS) OK 4
( 3 4 TOS) OK +
( 7 TOS)
```

```
def xbye():  "( --) Leave interpreter."; raise SystemExit
```

The `xbye` subroutine is called to halt the Forthish REPL (Read Eval Print Loop).

```
def xdot():  "( n --) Print TOS."; print(S.pop())
```

Print the top item on the Forthish stack and consume it (pop it from the stack).

Printout appears in the REPL, on the command line.

```
def xdots(): "( --) Print stack contents."; print(S)
```

Non-destructively print the full contents of the stack.

Useful for debugging and for understanding what is on the stack.

```
def xadd():  "( a b -- sum)"; S.append(S.pop() + S.pop())
```

Consume the top two items on the stack. Assume that they are integers. Add them together and push the result onto the stack.

```
def xswap(): "( a b -- b a)"; x = S[-1]; S[-1] = S[-2]; S[-2] = x
```

Swap the top two items on the stack. The second item down is moved to the top of the stack.

Semantically, this operates as

```
B ← top of stack, pop
A ← top of stack, pop (A used to be the 2nd item, but because of the
preceding pop, A has become the top)
push (B) onto the stack
push (A) on the stack
```

A is now the top of the stack and B is second.

In Python, this can be done more efficiently by indexing into the stack.

The top of the stack is `S[-1]` and the second item is `S[-2]`.

A temporary, local variable `x` is used to save the top item. The top slot is overwritten with the second item. Then, the saved value, `x` is written into the second slot.

```
def xsub():  "(a b -- diff)"; xswap(); S.append(S.pop() - S.pop())
```

Consume the top two items on the stack. Assume that they are integers. Subtract them (top - second, b - a) and push the result onto the stack.

```
def xword():
    "(char -- string) Read in string delimited by char."
    global BUFP
    want = chr(S.pop())
    found = ""
    while BUFP < len(BUFF):
        x = BUFF[BUFP]
        BUFP += 1
        if want == x:
            break
        else:
            found += x
    S.append(found)
```

Create a string containing the next word and push it onto the stack.

Parsing is controlled by a single-character delimiter. `Xword` copies characters from `BUFF` and increments `BUFP` until the delimiter is encountered or until `BUFP` points beyond the end of `BUFF`.

The single-character delimiter is at the top of the stack when this routine is entered. The delimiter is consumed (the stack is popped).

```
def xinterpret():
    "( string --) Execute word."
    word = S.pop()
    if not word:
        return
    if word in D:
        D[word]()
    elif word.isdigit():
        S.append(int(word))
    else:
        print(f"{word}?")
```

Consume the top item from the stack.

If the item is empty, ignore it and just return.

If the top item is a string and is found in the dictionary of words (subroutines) called `D`, then invoke the named subroutine. We fetch the subroutine by indexing the `D` dictionary. Since each word (subroutine) operates only on the Forthish stack, no parameters are passed to the subroutine.

If the top item is not found in the dictionary `D` *and* the word contains only digit characters, i.e. looks like a string representation of an integer, then convert it to a Python integer and push that integer onto the stack.

If the item doesn't match as above, then the interpreter doesn't know what to do with it - this is an error and we print a simple error message ("`...`?" where `...` is replaced by the string item that was popped from the stack).

```
D = {"bye": xbye, ".": xdot, ".s": xdots,
     "+": xadd, "swap": xswap, "-": xsub,
     "word": xword, "interpret": xinterpret}
```

This creates the subroutine dictionary `D` and initializes it with `key:value` pairs.

The `key` is the string representing a word.
The `value` is the Python function which implements the word.

For example `".": xdot` installs a word `.` (the single character `.`) and associates it with the Python subroutine `xdot`.

When the REPL interprets the word `.`, it will invoke the `xdot` subroutine.

```
def ok():
    "( --) Interaction loop -- REPL."
    global BUFF, BUFP
    while True:
        BUFF = input("OK ")
        BUFP = 0
        while BUFP < len(BUFF):
            S.append(32)  # ASCII space character.
            xword()
            xinterpret()
```

The main REPL for Forthish.

It loops reading input from the user, then looping again to interpret the input. Roughly:

```
loop
    get a line of input from user → BUFF
    loop
        parse one more word from BUFF
        interpret word
    end loop
end loop
```

Each line of input is parsed into its constituent words and those words are sequentially interpreted.

The interpretation is an inner loop that consists of the following steps:

- quit the inner loop when all characters from the last line of user input (`BUFF`) have been parsed
- push a *blank* (ASCII space) onto the stack, meant to be used as the delimiter
- call the subroutine `xword` to parse off the next word and to leave that word on the stack
- call the subroutine `xinterpret` to consume the top word and interpret it.

```
if "__main__" == __name__:
    ok()
```

This is the Python mainline. It invokes the REPL, called `ok`.