3. Some Simple Programs

The best way to learn Pyth is to dive head-first into it by writing some programs. We will be writing simple algorithmic exercises.

3.1. Factorial

An easy problem that will get you into thinking the correct way is computing factorials and all the different ways to do it. We will trying simple loops, the recursive approach, and the `reduce` function. Let's try it the obvious, straightforward way first. But how do we get input in Pyth?

3.1.1. Input in Pyth

There are many ways to get input in Pyth. The obvious one that handles all cases is the `w` function which acts exactly like Python's `raw_input`. In fact, it compiles to that. The preferred way over this is the variable `z` which is preinitialised to the input and is easier to use multiple times unlike `w`. But both it and `w` are string representations of the input and need to be evaluated with the `v` function. Hence, the most common input method is to use the `Q` variable which is preinitialised to the *evaluated* input so it can be used outright. Try using it:

```
input: 5

==================================================
*2Q
==================================================
Q=copy(literal_eval(input()))
Pprint("\n",times(2,Q))
==================================================
10
```

See? Easy.

Note: The parser checks if `Q` or `z` are used in the program and adds a line setting them to the input if they are used, as you can see from the debug output above.

3.1.2. For Loops In Pyth

Pyth has many of the control flow options available by Python, including `for` loops, which is what we will be using here. They are done by the `F` command and takes the name of a variable, the list to loop on, and then an infinite amount of code only bounded by a parenthesis or semicolon. Let's try looping from zero through ten:

```
==================================================
FNrZTN
==================================================
for N in Prange(0,T):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
```

Notice that:

1. The range function is `r`.
2. It is, like Python, not inclusive of the end value (We'll have to fix that).
3. Printing is implicit so we just had to name `N` and it was surrounded by a `Pprint` call.
4. `Z` is preinitialised to `0`

Oops! We forgot that like Python, range does not include the end value. We can do +1T, but Pyth's head function, `h`, also functions as an increment. Let's try that again:

```
==================================================
FNrZhTN
==================================================
for N in Prange(Z,head(T)):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
10
```

But we can make it shorter still. The `U` function, or "unary-range" is like Python's range with only one argument. It'll save one character by removing the need for the `Z`:

```
==================================================
FNUhTN
==================================================
for N in urange(head(T)):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
10
```

And even shorter. The `V` keyword is the "unary-range-loop" which does the looping through the one argument range. It uses N as the loop variable:

```
==================================================
VhTN
==================================================
for N in urange(head(T)):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
10
```

Notice the debug output for the last two were exactly the same. In fact, during preprocessing, the parser expands all occurrences of `V` to `FNU`.

Now we should be able to write an iterative factorial.

3.1.3. The Iterative Factorial

First, let's loop from one to the input. It should be easy, but we can't use `V` since we want our range to start from `1`, not `0`. Remember also to increment the input:

```
input: 5

==================================================
FNr1hQN
==================================================
Q=copy(literal_eval(input()))
for N in Prange(1,head(Q)):
 Pprint("\n",N)
==================================================
1
2
3
4
5
```

Now, we have to have our variable that holds the answer. Pyth has an assignment operator which works pretty much as you'd expect:

```
==================================================
=N5N
==================================================
N=copy(5)
Pprint("\n",N)
==================================================
5
```

But if you only need one variable, it's better to use `K` or `J` which don't need an equals sign to be assigned to on their first use:

```
==================================================
K5K
==================================================
K=5
Pprint("\n",K)
==================================================
5
```

Applying that to our factorial:

```
==================================================
K1FNr1hQ=K*KN
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
==================================================
```

Now we just have to print our answer which should be easy since it is implicit:

```
input: 5

==================================================
K1FNr1hQ=K*KNK
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
 Pprint("\n",K)
==================================================
1
2
6
24
120
```

Yikes! We forgot that a `for` loop's influence is infinite so the printing happens every time the loop runs. We can use a parenthesis since we only have to end one control flow, but it is better practice to use a semicolon:

```
input: 5

==================================================
K1FNr1hQ=K*KN;K
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
Pprint("\n",K)
==================================================
120
```

It works!

One final change we can make to shorten the program is to use Pyth's augmented assignment syntactic sugar. Just like Python has `+=`, `-=` and so forth, Pyth has the same constructs, except in reverse, such as `=+`, `=-`, etc. However, Pyth's augmented assignment can be used with any function, not just binary arithmetic operators. For instance, `=hK` has the same effect as `K++`. For this code, we will use `=*`:

```
input: 5

=================== 14 chars ====================
K1FNr1hQ=*KN;K
==================================================
assign('Q',eval(input()))
assign("K",1)
for N in num_to_range(Prange(1,head(Q))):
 assign('K',times(K,N))
 imp_print(K)
==================================================
120
```

3.1.4. User Defined Functions in Pyth

The most general way of defining functions in Pyth is with the `D` keyword. `D` works similarly to `def` in Python. To define a triple function called `h` that takes the input variable `Z`, you could write the following:

```
=================== 9 chars =====================
DhZK*3ZRK
==================================================
@memoized
def head(Z):
 K=times(3,Z)
 return K
==================================================
```

Note that `R` is the equivalent of `return`. Also, since arities in Pyth are unchangable, to define a new 1-variable function, an existing 1-variable function name must be used.

Pyth has a shorthand for function definition. They work similarly to lambdas in Python, in that there is an implicit return statement. The one var lambda uses the `L` keyword, uses the variable `b`, and defines a function named `y`.The two var lambda uses `M`, the variables `G` and `H`, and defines `g`.Here is a demonstration of a triple function:

```
==================================================
L*3b
==================================================
@memoized
def subsets(b):
 return times(3,b)
==================================================
```

And here's me calling it:

```
==================================================
L*3by12
==================================================
@memoized
def subsets(b):
 return times(3,b)
Pprint("\n",subsets(12))
==================================================
36
```

Note: All functions are automatically memoized in Pyth.

3.1.5. The Recursive Factorial

The recursive factorial is a common solution. It works by taking the factorial of the number lower than it, recursively, until you get to zero, which returns 1. Let's first define our factorial function's base case of zero:

```
=================== 5 chars =====================
L?bT1
==================================================
@memoized
def subsets(b):
 return (T if b else 1)
==================================================
```

Here I'm using `T` as a placeholder for the recursive case.

Also, notice that the ternary operator `?abc` evaluates to `if a then b else c`.

Now let's complete the factorial function:

```
=================== 9 chars =====================
L?b*bytb1
==================================================
@memoized
def subsets(b):
 return (times(b,subsets(tail(b))) if b else 1)
==================================================
```

This uses `t`, the decrement function, to recursively call the function on the input minus 1.

Pretty simple. Now we have to take input and run the function:

```
input: 5

=================== 11 chars ====================
L?b*bytb1yQ
==================================================
    assign('Q',literal_eval(input()))
    @memoized
    def subsets(b):
     return (times(b,subsets(tail(b))) if b else 1)
    imp_print(subsets(Q))
==================================================
120
```

Another factorial example...

3.1.6. Factorials With Reduce

The best way to do it, the way most people would do it, would be to use the reduce function. The `u` operator works exactly like Python's reduce, except for an implicit lambda so you can just write code without a lambda declaration. All a factorial is, is a reduction by the product operator on the range from 1 through n. This makes it very easy. The reduce function takes a statement of code, the sequence to iterate on, and a base case:

```
input: 5

==================================================
u*GHr1hQ1
==================================================
Q=copy(literal_eval(input()))
Pprint("\n",reduce(lambda G, H:times(G,H),Prange(1,head(Q)),1))
==================================================
120
```

As with each function in Pyth which uses an implicit lambda, reduce (`u`) has its own built in variables. In this case, the variables in question are `G`, the accumulator variable, and `H`, the sequence variable.

However, we can do better than this. If we use the list from `0` to `Q-1` instead, but multiply by one more than the sequence variable in the reduce, we can shorten the code. `UQ`, unary range of `Q` will produce the appropriate list, but `u` has a handy default where a number as the second variable will be treated identically to the unary range of that number. Thus, our code becomes:

```
input: 5

=================== 7 chars =====================
u*GhHQ1
==================================================
assign('Q',eval(input()))
imp_print(reduce(lambda G, H:times(G,head(H)),Q,1))
==================================================
120
```

Final way to calculate the factorial:

3.1.7. Factorial With Built-in

Pyth has a lot of specialty functions. So many, in fact, that there are too many to write them all with single character names. To remedy this, we use the `.` syntax. `.` followed by another character does something entirely different. `.!` in particular is the factorial function:

```
input: 5

=================== 3 chars =====================
.!Q
==================================================
assign('Q',eval(input()))
imp_print(factorial(Q))
==================================================
120
```

3.2. The First n Fibonacci numbers

The Fibonacci sequence is another subject of many programming problems. We will solve the simplest, finding the first n of them. The Fibonacci sequence is a recursive sequence where each number is the sum of the last two. It starts with 0 and 1. We'll just set the seed values and then loop through how many ever times needed, applying the rule. We will need a temp variable to store the previous value in the exchange:

```
input 10:

=================== 15 chars ====================
J1VQJKZ=ZJ=J+ZK
==================================================
    assign('Q',literal_eval(input()))
    assign("J",1)
    for N in num_to_range(Q):
     imp_print(J)
     assign("K",Z)
     assign('Z',J)
     assign('J',plus(Z,K))
==================================================
1
1
2
3
5
8
13
21
34
55
```

Notice that we used `Z` as one of the variables. `Z` is preinitialized to `0`, which was appropriate to use here. All of Pyth's variables have some sort of special property.

That was pretty easy, but this can be shortened (or should be) with the double-assignment operator, `A`. This has an arity of 1 and takes a tuple of two values. This shortens the assignment, but in this case we have to re-assign `H` and `G` since A implicitly uses them and their defaults are `{}` and an alphabetical string respectively. We use the `(` tuple creation operator to make the tuple:

```
input: 10

=================== 14 chars ====================
A(Z1)VQHA(H+HG
==================================================
    assign('Q',literal_eval(input()))
    assign('[G,H]',Ptuple(Z,1))
    for N in num_to_range(Q):
     imp_print(H)
     assign('[G,H]',Ptuple(H,plus(H,G)))
==================================================
1
1
2
3
5
8
13
21
34
55
```

We will examine some more exercises in the next chapter.