Contents

Deoxyribose **1**
Introducing Deoxyribose . . . 1
Amino acids (operators) . . . 1
Special codons . . . 1
Charged amino acids — Single-stack operations . . . 2
Non-polar amino acids — Two-stack operations . . . 2
Polar amino acids — Flow control . . . 2
Integer literals . . . 2
Examples . . . 3
Hello, world! . . . 3
Infinite Fibonacci sequence . . . 3
Cat . . . 4
Print integers from 1 to N . . . 4
Primality test . . . 5
Truth machine . . . 6
Notes etc. . . . 7

Deoxyribose

The DNA-themed programming language all the kids are talking about

Introducing Deoxyribose

Deoxyribose is a stack-based esoteric programming language based on the syntax and function of DNA.

A valid deoxyribose program is made up entirely of the letters A, C, G, and T, which form three-digit **codons** in accordance with the DNA codon table. These codons correspond to stack or flow-control operations. All other characters are ignored, making the language suitable for polyglots and other such fun things.

Interestingly, the correspondence between codons and operations is **degenerate**, so there's more than one way to write each operator, just like in the real genetic code.

Execution of Deoxyribose takes place on a closed circle of DNA (like the bacterial genome). Any amount of information can be placed before the first start codon, and if no stop codon is found, execution will loop around to the very beginning. Clever use of the frameshifts this can lead to, combined with the degeneracy of the codon–amino acid correspondence, can allow some very fun and exciting things.

The name of the language was selected due to being the "most-DNA-y" word that doesn't contain an A, C, G, or T, meaning it can be included in a comment or shebang (`#! /usr/bin/env deoxyribose` is a valid comment).

Amino acids (operators)

The operators are defined by their correspondence to particular amino acids, as follows.

Special codons

- **Start** (`ATG`): Start program execution
- **Stop** (`TAG`, `TAA`, and `TGA`): *End* program execution & return 0

Charged amino acids — Single-stack operations

- **His**: *Push* next codon (expressed in quaternary notation, see below) to the top of the main stack
- **Lys**: *Pop* top of main stack to standard output as a number
- **Arg**: *Pop* top of main stack to standard output as a Unicode character
- **Glu**: *Dupe* (duplicate) the top element of the main stack
- **Asp**: *Drop* the top element of the main stack

Non-polar amino acids — Two-stack operations

- **Leu**: *Plus* (add) the top elements of the main and auxiliary stacks, remove these elements, and place the result on top of the main stack
- **Ile**: *Minus* (subtract) the top element of the auxiliary stack from the top element of the main stack, remove these elements, and place the result on top of the main stack
- **Val**: *Mul* (multiply) the top elements of the main and auxiliary stacks, remove these elements, and place the result on top of the main stack
- **Pro**: *Divide* the top element of the main stack by the top element of the auxiliary stack, round the result towards zero, remove these elements, and place the result on top of the main stack
- **Met**: *Swop* (swap) the top elements of the main and auxiliary stacks
- **Phe**: *Join* (concatenate) the main and auxiliary stacks by placing the auxiliary stack on top of the main stack, leaving the auxiliary stack empty
- **Gly**: *Move* the top element of the main stack to the top of the auxiliary stack
- **Trp**: *Power* (exponentiate) the top element of the main stack to the power of the top element of the auxiliary stack, remove these elements, and place the result on top of the main stack
- **Ala**: *Modulo* — calculate the remainder when the top element of the main stack is divided by the top element of the auxiliary stack, remove both of these elements, and place the result on top of the main stack

Polar amino acids — Flow control

- **Cys**: *Jump* – Jump to the *next* occurrence of the next codon
- **Asn**: *Loop* – Jump backwards to the *previous* occurrence of the next codon
- **Ser**: *Jump if <=0* – If the top element of the main stack is less than or equal to zero, jump to the *next* occurrence of the next codon
- **Thr**: *Loop if <=0* – If the top element of the main stack is less than or equal to zero, jump to the *previous* occurrence of the next codon
- **Tyr**: *Jump if null* – If the main stack is empty, jump to the *next* occurrence of the next codon
- **Gln**: *Loop if null* – If the main stack is empty, jump to the *previous* occurrence of the next codon

Note that the length of these jumps need not always be a multiple of three. This can cause frameshifts, which are half the fun.

Integer literals

Integer literals are expressed as a three-digit number in base-4, such that `A` = 0, `C` = 1, `G` = 2, and `T` = 3. This limits integer literals to the range `AAA` (0) to `TTT` (63).

Once stored inside the stack, values are not subject to these same limits, and are treated just like ordinary Python numbers. Larger values, floats, and negative numbers can therefore be constructed using mathematical operations.

Unicode characters can be stored as their character reference and converted back by arginine. There is no built-in string (or array) datatype, nor any real distinction between ints and floats, only numbers ordered on the stack.

Examples

Note that the examples below include spaces to separate segments of the code for readability; these are unnecessary, ignored by the interpreter, and excluded from the given byte counts.

The multi-line explanations given below each example will not run as expected unless all A, C, G, and T characters (case-insensitive) are removed from the comments; all other comment characters are fine.

These examples generally implement a naïve and accessible approach to a problem, in order to demonstrate the power and concept of programming in Deoxyribose. These solutions may be suboptimal in terms of performance and byte count. Golfing them down is left as an exercise to the reader (but the reader is encouraged to submit shorter solutions as pull requests).

Hello, world!

`ATG CATGAC CACTTTCACGCCGGTTTA ... CATTTTCATAGCGGTTTG AGATATATTAATTTG` (Truncated because it's long and boring, actually prints `Hd!`)

Accepts no input; prints `Hello, world!` to STDOUT.

```
ATG                                 TGA End

CAT His     Push
GAC 33      (ASCII !)

CAC His     Push
TTT 63
CAC His     Push
GCC 37
GGT Gly     Move
TTA Leu     Plus (= 100, ASCII d)

...

CAT His     Push
TTT 63
CAT His     Push
AGC 9
GGT Gly     Move
TTG Leu     Plus (= 72, ASCII H)

AGA Arg     Pop as char
TAT Tyr     Jump if null
ATT
AAT Asn     Loop
TTG                                 ATT
```

Infinite Fibonacci sequence

`ATG CATAACGAA GGT GAATTAGGCATGGAAAAAAATGGT` (39 B)

Accepts no input; prints an infinite series of newline-separated integers to STDOUT. The printed sequence starts at 2, but it would be trivial to add the expected `1\n1\n` at the expense of a few more bytes.

```
ATG

CCT Pro     Divide (yields 1)
GAA Glu     Dupe

GGT Gly     Move

GAA Glu     Dupe
TTA Leu     Plus
GGC Gly     Move
ATG Met     Swop
GAA Glu     Dupe
AAA Lys     Pop
AAT Asn     Loop
GGT "
```

Cat

`ATG GGTTATTGTAATATGT TTT AGATATTCTAATTTTCTTA` (41 B)

Accepts any number of characters or Unicode codepoints as arguments; prints its input to the screen. If the input contains spaces, these will be stripped unless the string is wrapped in quotes. Input containing numbers is fine, but entirely numeric input will be converted into the corresponding Unicode character (e.g. input of `100` prints `d`).

```
ATG

GGT Gly     Move
TAT Tyr     Jump if null
TGT "
AAT Asn     Loop
ATG "
T

TTT Phe     Join

AGA Arg     Pop as char
TAT Tyr     Jump if null
TCT "
AAT Asn     Loop
TTT "
CT
TA
```

Print integers from 1 to N

`ATG GGTCATAACGAAGGTCCT GAAAAACATAACGGTTTATTTGAAGGTGGT GAAATTAGTTAG TAGGATAATCCT` (75 B)

Accepts an integer N as input; prints all integers from 1 to N to STDOUT, separated by newlines.

```
ATG
```

```
GGT Gly     Move
CAT His     Push
AAC 1
GAA Glu     Dupe
GGT Gly     Move
CCT Pro     Divide

GAA Glu     Dupe
AAA Lys     Pop
CAT His     Push
AAC 1
GGT Gly     Move
TTA Leu     Plus
TTT Phe     Join
GAA Glu     Dupe
GGT Gly     Move
GGT Gly     Move

GAA Glu     Dupe
ATT Ile     Minus
AGT Ser     Jump if <= 0
TAG

TAG End

GAT Asp     Drop
AAT Asn     Loop
CCT
```

Primality test

`ATG GAACATAAG GAGGGTGGC GCT CATAACGGT AGTGAC GATGAATTTGGTTTA AATAAG GAAGAC GATTTTGATGGTATT AGTTAG CATAAAAAATAG CATAACAA` (107 B)

Accepts one integer greater than 1 as input; prints 1 if prime, 0 if composite.

```
ATG Start

GAA Glu     Dupe
CAT His     Push
AAG 2

GAG Glu     Dupe
GGT Gly     Move
GGC Gly     Move

GCT Ala     Modulo

CAT His     Push
AAC 1
GGT Gly     Move

AGT Ser     Jump if <= 0
GAC "
```

```
AAA Lys     Pop

TGG End
```

```
GAT Asp     Drop

GAA Glu     Dupe
TTT Phe     Join
GGT Gly     Move
TTA Leu     Plus

AAT Asn     Loop
AAG "

GAA Glu     Dupe
GAC Asp     Drop

GAT Asp     Drop
TTT Phe     Join
GAT Asp     Drop
GGT Gly     Move
ATT Ile     Minus

AGT Ser     Jump if <= 0
TAG "

CAT His     Push
AAA 0
AAA Lys     Pop
TAG Stop

CAT His     Push
AAC 1
AA
```

Truth machine

`ATG GAG AAG AGC ATA AAT` (18 B)

Accepts one value as input. If this value is less than or equal to 0, it is printed once before the program terminates. If the value is greater than 0, it is printed infinitely.

```
ATG                         ATA
GAG Glu     Dupe            TGG Trp     Power
AAG Lys     Pop             AGA Arg     Unipop
AGC Ser     Jump if <= 0    AGA Arg     Unipop
ATA "                       GCA Ala     Modulo
AAT Asn     Loop            TAA End
```

- Execution starts at the initial ATG.
- The top element of the main stack is duplicated & printed, then
  - If the value is less than or equal to zero, jump to the ATA formed by the Asn and the start codon (remember, the code is circular), then run through a series of no-ops, including printing some non-printable characters.
  - If the value is greater than 0, loop back to the start.

If you don't like the ugly sequence of no-ops, adding `ATG` to the end to make the loop target explicit results in a clean termination immediately after the jump, for three more bytes.

Notes etc.

This is very much a work in progress, and some aspects of the design will probably be changed in backwards-incompatible ways in future versions. I would welcome suggestions.

The specification and documentation for this project are dual-licensed under a CC-BY licence and the MIT licence (see the `LICENSE` file), and the interpreter etc. are under an MIT licence. I'd love to see them developed and improved upon, but it would be nice to see my name on there somewhere.