Python I: Values, variables and strings

In the UNIX lectures you ran programs, glued them together with pipes, and wrote bash scripts with variables, `if` and loops. Python is the next tool: a general-purpose programming language that is easy to read, has a huge collection of add-on packages for science (Biopython, pandas, numpy, matplotlib), and is one of the most-used languages in bioinformatics.

What you'll learn

- what Python is and when to use it instead of bash
- three ways to run Python code: the interactive interpreter, scripts, and notebooks
- the basic kinds of values (*types*): integers, decimals, text, `True`/`False`, `None`
- variables, arithmetic, and converting between types
- strings: indexing, slicing and string methods - the tools for working with DNA and protein sequences
- printing results with f-strings
- how to read an error message (a *traceback*) and fix the problem

By the end you will be able to compute the GC content and reverse complement of a DNA sequence and print a nicely formatted result.

Python and bash

Both bash and Python are *interpreted*: you write the code in a text file and a program (the *interpreter*) reads it and runs it line by line. You don't compile anything.

Use bash when ...	Use Python when ...
you are running other programs (BLAST, samtools)	you are doing calculations or complicated logic
looping over files and submitting jobs	parsing a file format (FASTA, GFF, VCF)
a quick `grep`/`cut`/`sort`/`awk` pipeline works	you need to keep data in memory and look it up
moving, renaming and compressing files	you want statistics, tables (pandas) or plots

In practice you will use both: a bash script that runs a Python script on every sample, or a Python script whose output you pipe into `sort` and `head`.

There were two major versions of Python. Python 2 ended in 2020; everything in this course is **Python 3**. If you see `print "hello"` (without parentheses) online, it is old Python 2 code.

Running Python

Is Python installed?

On the cluster (and on most Macs and Linux computers) the command is `python3`:

```
python3 --version
```

```
Python 3.12.14
```

Your version may be different; anything 3.9 or newer works for these lectures. (The outputs in these notes were made with Python 3.12.) In Python IV we set up a conda environment with a newer Python and the packages we need. On some systems `python` (no 3) is also Python 3; on others it doesn't exist. Use `python3` to be safe.

The interactive interpreter

Type `python3` by itself to start an interactive session. The `>>>` prompt means Python is waiting for you to type something. It runs each line as soon as you press Enter and shows you the result (the version line you see will be different):

```
python3
Python 3.12.14 (main, Sep  1 2026, 14:09:38) [Clang 22.1.3 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>> print("hello world")
hello world
>>> exit()
```

Leave with `exit()` or Ctrl-D. The interpreter is a great calculator and a place to try things out. In these notes, examples that start with `>>>` show an interactive session: type the part after `>>>` and Python prints the line below it. Examples that follow each other in the same section continue the same session, so a variable made in one example is still there in the next.

Scripts

For anything you want to keep or rerun, write a **script**: a text file ending in `.py`. Make a file called `hello.py` (with `nano hello.py`, or the editor in Jupyter or VS Code) that contains this one line:

```
print("Hello World!")
```

Then run it by giving the file name to `python3` - just like `bash myscript.sh`:

```
python3 hello.py
```

```
Hello World!
```

Output from `print` goes to standard output (STDOUT), so everything you learned about redirection and pipes works:

```
python3 hello.py > message.txt
cat message.txt
```

```
Hello World!
```

Making a script executable

As with bash scripts, you can add a **shebang** line at the top so the script can be run by name. `#!/usr/bin/env python3` means "find `python3` on my PATH and use it". Save this as `hello2.py`:

```
#!/usr/bin/env python3
# hello2.py - my first Python script
print("Hello from a script!")
```

```
chmod +x hello2.py
./hello2.py
```

```
Hello from a script!
```

The shebang must be the very first line. Lines starting with `#` are **comments**: Python ignores everything from `#` to the end of the line. Use comments to explain *why* the code does something.

Jupyter notebooks and VS Code

Through OnDemand you can start a Jupyter notebook or VS Code on the cluster in your web browser. A notebook is made of *cells*; you type code in a cell and press Shift-Enter to run it, and the result of the last line is shown below the cell (like the `>>>` prompt). Notebooks are good for exploring, but homework is turned in as `.py` scripts, so practice both. Everything in these notes works in either place.

Values and types

Every value in Python has a **type**, which decides what you can do with it. The five basic types are:

Type	What it holds	Examples
`int`	whole numbers	`42`, `-7`, `3000000`
`float`	decimal numbers	`0.52`, `3.14`, `1e-5`
`str`	text (a *string*)	`"ATGC"`, `'chrI'`, `"42"`
`bool`	true or false	`True`, `False`
`None`	"nothing", no value	`None`

The `type()` function tells you the type of a value:

```
>>> type(42)
<class 'int'>
>>> type(0.52)
<class 'float'>
>>> type("ATGC")
<class 'str'>
>>> type("42")
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```

Notice that `"42"` in quotes is a *string*, not a number. This matters a lot: every line you read from a file is a string, even if it looks like a number, and you have to convert it before doing math (see Converting between types).

`1e-5` is scientific notation for 0.00001 (you will see it in BLAST E-values). `True`, `False` and `None` must be capitalized exactly like that.

Variables

A **variable** is a name that refers to a value. Create one with `=` (the *assignment* operator). Unlike bash, spaces around `=` are fine and you don't need `$` to use the variable:

```
>>> seq = "ATGGCGTACGCTTAG"
>>> gene_name = "YFG1"
>>> length = 15
>>> seq
'ATGGCGTACGCTTAG'
>>> length * 2
30
```

In the interpreter, typing a variable name shows its value. In a script you have to `print()` it.

A variable can be given a new value at any time, even one of a different type. The *augmented assignment* operators `+=`, `-=`, `*=` and `/=` update a variable using its current value, which is handy for counting:

```
>>> count = 0
>>> count += 1
>>> count += 1
>>> count
2
```

`count += 1` means exactly the same as `count = count + 1`.

Naming rules. Names can contain letters, digits and `_`, but can't start with a digit. They are case-sensitive (`Seq` and `seq` are different variables). By convention Python code uses lowercase words joined with underscores (`gc_content`, `num_genes`). Choose names that say what the value is: `exon_length` is much easier to read later than `x`. Don't use names that Python already uses for something, such as `str`, `list`, `len`, `sum` or `type` - it works, but then the built-in function stops working in your program.

A variable doesn't exist until you assign to it. Using it before that is a `NameError`:

```
>>> print(total_length)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'total_length' is not defined
```

Numbers and arithmetic

Operator	Meaning	Example	Result
`+`	add	`7 + 2`	`9`
`-`	subtract	`7 - 2`	`5`
`*`	multiply	`7 * 2`	`14`
`/`	divide (always gives a float)	`7 / 2`	`3.5`
`//`	integer (floor) division	`7 // 2`	`3`
`%`	remainder (modulo)	`7 % 2`	`1`
`**`	power	`7 ** 2`	`49`

```
>>> 10 / 2
5.0
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 10 % 3
1
>>> 2 ** 10
1024
```

`/` always gives a `float`, even when the answer is a whole number (`5.0`). Use `//` when you want a whole number.

`//` and `%` come up all the time with sequences. A coding sequence should have a length that is a multiple of 3; `%` tells you the remainder:

```
>>> cds_length = 1236
>>> cds_length // 3
412
>>> cds_length % 3
0
>>> 1237 % 3
1
```

So this CDS has 412 codons and nothing left over; a length of 1237 would leave one extra base. `//` is also how you put values into *bins*. To find which 500 bp bin a gene of length 1812 falls into, divide by the bin size, drop the remainder, and multiply back:

```
>>> 1812 // 500
3
>>> 1812 // 500 * 500
1500
```

Order of operations is the usual math order: `**` first, then `*`, `/`, `//`, `%`, then `+` and `-`. Operators at the same level go left to right. Use parentheses whenever you are unsure - they make the code easier to read too:

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
>>> g = 21
>>> c = 19
>>> total = 80
>>> g + c / total * 100
44.75
>>> (g + c) / total * 100
50.0
```

The first GC calculation is wrong because `/` and `*` happen before `+`.

Decimals are not exact. Computers store floats in binary, so some decimal numbers can't be stored exactly:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> round(0.1 + 0.2, 2)
0.3
```

This is normal and not a bug in your code. Round when you print a result (see f-strings), and don't test floats for exact equality.

Some other useful built-in functions for numbers are `abs()`, `round()`, `min()` and `max()`:

```
>>> abs(-5)
5
>>> round(3.14159, 2)
3.14
>>> min(1812, 750, 2210)
750
>>> max(1812, 750, 2210)
2210
```

Converting between types

`int()`, `float()` and `str()` convert a value to another type:

```
>>> int("1812")
1812
>>> float("0.52")
0.52
>>> str(1812)
'1812'
>>> int(3.99)
3
>>> float(7)
7.0
```

`int()` of a float throws away the decimal part (it does not round). Why do we need this? Because text read from a file is always a string. In a BED file the start and end of a feature are read as the strings `"21408673"` and `"21408826"`. Adding two strings *joins* them rather than doing arithmetic, and subtracting them is an error:

```
>>> start = "21408673"
>>> end = "21408826"
>>> start + end
```

```
'2140867321408826'
>>> end - start
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> int(end) - int(start)
153
```

Converting only works if the text really is a number:

```
>>> int("12.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.5'
>>> int(float("12.5"))
12
```

Strings

A **string** (`str`) is a sequence of characters: a DNA sequence, a gene name, a line from a file. Most of what we do with biological data is string processing.

Quotes and special characters

Make a string with single `'...'` or double `"..."` quotes; they mean the same thing. Pick the one that lets you put the other kind inside:

```
>>> "ATGC"
'ATGC'
>>> 'ATGC'
'ATGC'
>>> "5' untranslated region"
"5' untranslated region"
```

A backslash `\` starts an **escape sequence** for characters you can't type directly. The two you will use constantly are `\t` (tab) and `\n` (newline):

Escape	Meaning
`\t`	tab
`\n`	newline
`\\`	a single backslash
`\' \"`	a quote inside the same kind of quotes

```
print("gene\tlength")
print("YFG1\t1812")
print("line one\nline two")
```

```
gene    length
YFG1    1812
line one
line two
```

Triple quotes (`"""..."""`) make a string that can span several lines; they are often used for long comments at the top of a script.

Joining and repeating strings

`+` joins (*concatenates*) strings, and `*` repeats a string:

```
>>> "chr" + "I"
'chrI'
>>> "ATG" + "GCG" + "TAA"
'ATGGCGTAA'
>>> "N" * 10
'NNNNNNNNNN'
>>> "-" * 20
'--------------------'
```

`+` only works if both sides are strings. `"chr" + 1` is a `TypeError`; write `"chr" + str(1)` instead (or use an f-string, below).

Length, indexing and slicing

`len()` gives the number of characters in a string:

```
>>> seq = "ATGGCGTACGCTTAG"
>>> len(seq)
15
```

Each character has a position number called its **index**. Python counts from **0**, so the first character is `seq[0]`. Negative indexes count back from the end: `seq[-1]` is the last character.

```
index:       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
base:        A   T   G   G   C   G   T   A   C   G   C   T   T   A   G
negative:  -15 -14 -13 -12 -11 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1
```

```
>>> seq[0]
'A'
>>> seq[3]
'G'
>>> seq[-1]
'G'
>>> seq[-3]
'T'
```

A **slice** `seq[start:end]` gives the part of the string from index `start` up to *but not including* `end`. The length of a slice is `end - start`. Leave out `start` to begin at the start, or `end` to go to the end:

```
>>> seq[0:3]
'ATG'
>>> seq[3:6]
'GCG'
>>> seq[:3]
'ATG'
>>> seq[-3:]
'TAG'
>>> seq[3:]
'GCGTACGCTTAG'
```

So `seq[0:3]` is the start codon and `seq[-3:]` is the stop codon. The n-th codon (counting from 0) is `seq[3*n : 3*n+3]`:

```
>>> n = 2
>>> seq[3*n : 3*n+3]
'TAC'
```

A third number in a slice is the **step**. `seq[::3]` takes every third base, and a step of `-1` walks backward, which reverses the string:

```
>>> seq[::-1]
'GATTCGCATGCGGTA'
```

Asking for a single index past the end is an `IndexError`, but slices past the end are allowed and just stop at the end:

```
>>> seq[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> seq[10:100]
'CTTAG'
```

Genome coordinates and slices. BED files use the same "start at 0, end not included" convention as Python, so a BED feature `chrI 100 200` is exactly `chrom_seq[100:200]`. GFF and VCF files count from **1** and include the end, so a GFF feature from 101 to 200 is `chrom_seq[100:200]` - subtract 1 from the start only. Off-by-one errors like this are among the most common bugs in bioinformatics.

Strings can't be changed

Strings are **immutable**: once made, you can't change a character in place:

```
>>> seq[0] = "T"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Instead, build a new string and (if you want) assign it back to the same name:

```
>>> seq = "T" + seq[1:]
>>> seq
'TTGGCGTACGCTTAG'
```

String methods

A **method** is a function that belongs to a value; you call it with a dot: `value.method()`. Methods never change the original string (they can't - it's immutable); they *return* a new value. These are the ones you'll use most:

Method	What it does
`s.upper()`, `s.lower()`	upper- or lowercase copy
`s.strip()`	remove spaces, tabs and newlines from both ends
`s.count(sub)`	how many times `sub` occurs (non-overlapping)
`s.find(sub)`	index of the first `sub`, or `-1` if not found
`s.replace(old, new)`	copy with every `old` replaced by `new`
`s.startswith(prefix)`	`True` if `s` starts with `prefix` (also `endswith`)
`s.split(sep)`	break into a list of pieces at each `sep`
`sep.join(pieces)`	glue a list of strings together with `sep`

```
>>> dna = "atgGCGtaaNNacgt"
>>> dna.upper()
'ATGGCGTAANNACGT'
>>> dna
'atgGCGtaaNNacgt'
>>> dna = dna.upper()
>>> dna.count("A")
```

```
4
>>> dna.count("N")
2
>>> dna.find("TAA")
6
>>> dna.find("TGA")
-1
>>> dna.replace("N", "")
'ATGGCGTAAACGT'
>>> dna.startswith("ATG")
True
```

Notice that `dna.upper()` did not change `dna` until we assigned the result back.

`strip()` removes *whitespace* (spaces, tabs, newlines) from the ends. Every line you read from a file ends with a newline character, so you will call `strip()` on almost every line. The `repr()` function shows a string with its hidden characters visible:

```
>>> line = "  ATGGCG\n"
>>> line.strip()
'ATGGCG'
>>> print(repr(line))
'  ATGGCG\n'
```

`split()` turns a string into a **list** of strings (lists are covered in Python II). This is how we pull the columns out of a line of a tab-delimited file. With no argument, `split()` splits on any run of whitespace:

```
>>> bed_line = "Chr7\t21408673\t21408826"
>>> bed_line.split("\t")
['Chr7', '21408673', '21408826']
>>> "YFG1  1812   wild type".split()
['YFG1', '1812', 'wild', 'type']
```

Put `[ ]` after the split to take one piece. Here is the last column of a GFF line, which holds `key=value` pairs separated by `;`:

```
>>> attributes = "ID=YAL069W;Name=YAL069W;orf_classification=Dubious"
>>> attributes.split(";")
['ID=YAL069W', 'Name=YAL069W', 'orf_classification=Dubious']
>>> attributes.split(";")[0]
'ID=YAL069W'
>>> attributes.split(";")[0].split("=")[1]
'YAL069W'
```

`join()` is the opposite of `split()`. It is called on the *separator* and joins a list of strings:

```
>>> "\t".join(["YFG1", "chrI", "1812"])
'YFG1\tchrI\t1812'
>>> "-".join(["ATG", "GCG", "TAA"])
'ATG-GCG-TAA'
```

Methods can be **chained**: each one works on the result of the one before, left to right:

```
>>> "  atgcNNatgc\n".strip().upper().replace("N", "")
'ATGCATGC'
```

There are many more methods; see the Python documentation on string methods.

Printing results and f-strings

`print()` writes its arguments to STDOUT separated by spaces and followed by a newline. You can change the separator with `sep=` and the ending with `end=`:

```
print("YFG1", 1812, 0.52)
print("YFG1", 1812, 0.52, sep="\t")
print("no newline here", end="")
print(" - so this continues the same line")
```

```
YFG1 1812 0.52
YFG1    1812    0.52
no newline here - so this continues the same line
```

The best way to build output is an **f-string**: put an `f` before the opening quote, and anything inside `{ }` is replaced by its value. You can put any expression inside the braces, not just a variable name:

```
gene = "YFG1"
length = 1812
print(f"{gene} is {length} bp long")
print(f"{gene} has {length // 3} codons")
```

```
YFG1 is 1812 bp long
YFG1 has 604 codons
```

After a `:` inside the braces you can give a **format**: the number of decimal places, a percentage, or a column width. This is how you avoid printing `0.30000000000000004`:

```
gc = 7 / 15
print(gc)
print(f"GC = {gc:.2f}")
print(f"GC = {gc:.1%}")
print(f"{'gene':<8}|{'length':>8}|")
print(f"{'YFG1':<8}|{1812:>8}|")
print(f"{3000000:,} bp")
```

```
0.4666666666666667
GC = 0.47
GC = 46.7%
gene    |  length|
YFG1    |    1812|
3,000,000 bp
```

Format	Meaning	`f"{x:...}"` with `x = 1234.5678`
`.2f`	2 decimal places	`1234.57`
`.0f`	no decimals (rounded)	`1235`
`.1%`	multiply by 100, add `%`, 1 decimal	`123456.8%`
`.2e`	scientific notation	`1.23e+03`
`,`	thousands separators	`1,234.5678`
`>10`	right-align in 10 characters	`' 1234.5678'`
`<10`	left-align in 10 characters	`'1234.5678 '`

To make tab-delimited output that other programs (or `sort`, `cut`, R) can read, put `\t` between the fields:

```
gene = "YFG1"
chrom = "chrI"
length = 1812
gc = 0.4234
print(f"{gene}\t{chrom}\t{length}\t{gc:.3f}")
```

```
YFG1    chrI    1812    0.423
```

Older styles. You will see two other ways of formatting in older code and online examples. They do the same thing as the f-string above; you don't need to write them:

```
print("%s is %d bp, GC=%.2f" % ("YFG1", 1812, 0.4234))
print("{} is {} bp, GC={:.2f}".format("YFG1", 1812, 0.4234))
```

```
YFG1 is 1812 bp, GC=0.42
YFG1 is 1812 bp, GC=0.42
```

Putting it together: GC content and reverse complement

GC content is the fraction of bases that are G or C. With `count()` and `len()` it is one line:

```
seq = "ATGGCGTACGCTTAGCCGATAA"
gc = (seq.count("G") + seq.count("C")) / len(seq)
print(f"length={len(seq)} GC={gc:.3f}")
```

```
length=22 GC=0.500
```

The **reverse complement** is the sequence of the other strand read 5' to 3': swap A<->T and C<->G, then reverse. A first idea is to use `replace()` for each base, but that doesn't work:

```
seq = "AACG"
print(seq.replace("A", "T").replace("T", "A"))
```

```
AACG
```

After the first `replace`, the new `T`s can't be told apart from the original ones, so the second `replace` turns them all back into `A`. The right tool is `translate()`, which swaps every character at once using a table made with `str.maketrans()`: each character in the first string is replaced by the character at the same position in the second string.

```
seq = "ATGGCGTACGCTTAG"
complement = seq.translate(str.maketrans("ACGT", "TGCA"))
revcomp = complement[::-1]
print(seq)
print(complement)
print(revcomp)
```

```
ATGGCGTACGCTTAG
TACCGCATGCGAATC
CTAAGCGTACGCCAT
```

Here is everything as a script, `seq_report.py`. Note the comments and the descriptive variable names:

```
#!/usr/bin/env python3
# seq_report.py - print a short report about one DNA sequence

gene = "YFG1"
seq = "atgGCGTACGCTTAGccgataaNNN"

# clean up: uppercase, and drop the Ns (unknown bases)
seq = seq.upper().replace("N", "")

length = len(seq)
gc = (seq.count("G") + seq.count("C")) / length
revcomp = seq.translate(str.maketrans("ACGT", "TGCA"))[::-1]

print(f"gene:       {gene}")
```

```
print(f"length:     {length} bp ({length // 3} codons, {length % 3} left over)")
print(f"start:      {seq[:3]}  stop: {seq[-3:]}")
print(f"GC content: {gc:.1%}")
print(f"A={seq.count('A')} C={seq.count('C')} G={seq.count('G')} T={seq.count('T')}")
print(f"revcomp:    {revcomp}")
```

```
python3 seq_report.py
```

```
gene:       YFG1
length:     22 bp (7 codons, 1 left over)
start:      ATG  stop: TAA
GC content: 50.0%
A=6 C=5 G=6 T=5
revcomp:    TTATCGGCTAAGCGTACGCCAT
```

Inside the f-string we used single quotes (`seq.count('A')`) because the f-string itself uses double quotes.

Reading error messages

Everyone's code has errors, all the time. The skill is reading the message Python gives you. When something goes wrong Python stops and prints a **traceback**. Here is a script with a mistake in it:

```
seq = "ATGGCGTACGCTTAG"
length = len(seq)
print("Length is " + length)
```

```
python3 broken.py
```

```
Traceback (most recent call last):
  File "/rhome/yourname/python1/broken.py", line 3, in <module>
    print("Length is " + length)
          ~~~~~~~~~~~~~~^~~~~~~~
TypeError: can only concatenate str (not "int") to str
```

Read a traceback **from the bottom up**:

1. The last line is the most important: the **type** of error (`TypeError`) and a description: you can only join a string (`str`) to another string, not to an `int`.
2. Above it is the **file** (with its full path) and the **line number** (`line 3`), then the line itself. Python 3.11 and newer also underline the part that failed.
3. Fix: `print("Length is " + str(length))`, or better `print(f"Length is {length}")`.

The errors you will see most often:

Error	Usually means	Example
`SyntaxError`	Python can't understand the line: a missing quote, bracket or `:`	`print("ATG)`
`IndentationError`	the spaces at the start of a line are wrong	an indent for no reason
`NameError`	a typo, or a variable used before it was assigned	`pirnt(seq)`
`TypeError`	an operation on the wrong type	`"chr" + 1`
`ValueError`	the right type but a bad value	`int("12.5")`
`IndexError`	an index past the end of a string or list	`"ATG"[3]`
`AttributeError`	that type has no such method	`(1812).upper()`

A `NameError` often suggests the name you probably meant:

```
sequence = "ATGGCGTACGCTTAG"
print(len(sequnce))
```

```
python3 typo.py
```

```
Traceback (most recent call last):
  File "/rhome/yourname/python1/typo.py", line 2, in <module>
    print(len(sequnce))
              ^^^^^^^
NameError: name 'sequnce' is not defined. Did you mean: 'sequence'?
```

A `SyntaxError` is found *before* the program runs at all, so nothing is printed except the error. The line number is where Python noticed the problem, which is sometimes the line *after* the real mistake (for example, a missing closing parenthesis):

```
print("start")
print("GC is", 0.5
print("done")
```

```
python3 syntax.py
```

```
  File "/rhome/yourname/python1/syntax.py", line 2
    print("GC is", 0.5
          ^
SyntaxError: '(' was never closed
```

Tips for fixing errors:

- Fix the **first** error first, then run again.
- Check the line reported *and the line before it.*
- `print()` the variables involved and their `type()` just before the line that fails.
- Copy the last line of the message into a search engine; someone has had it before.

(Older versions of Python give shorter messages without the `^^^` markers or the "Did you mean" hints, but the last line is the same.)

Common mistakes

- **Counting from 1.** The first character is `seq[0]`, and `seq[1:4]` is the 2nd to 4th characters.
- **Forgetting the end of a slice is not included.** `seq[0:3]` has 3 characters, indexes 0, 1 and 2.
- **Doing math on strings from a file.** `"100" + "50"` is `'10050'`. Convert with `int()` or `float()` first.
- **Expecting a method to change the string.** `seq.upper()` on its own line does nothing useful; write `seq = seq.upper()`.
- **Mixing quote types.** `"5' UTR'` is not a complete string. Start and end with the same kind of quote.
- **Case matters.** `print` not `Print`; `True` not `true`; `seq.count("a")` does not count `"A"`.
- **Missing parentheses.** In Python 3, `print` is a function: `print("hi")`, not `print "hi"`.

Quick reference

Task	Code
run a script	`python3 script.py`
shebang line	`#!/usr/bin/env python3`
comment	`# this is ignored`
what type is it?	`type(x)`
convert	`int("42")`, `float("0.5")`, `str(42)`
integer division, remainder	`n // 3`, `n % 3`
power, rounding	`2 ** 10`, `round(x, 2)`
length	`len(seq)`
first, last character	`seq[0]`, `seq[-1]`
slice (end not included)	`seq[3:6]`, `seq[:3]`, `seq[-3:]`

Task	Code
reverse	`seq[::-1]`
upper/lower case	`seq.upper()`, `seq.lower()`
remove whitespace at ends	`line.strip()`
count, find	`seq.count("G")`, `seq.find("ATG")`
replace	`seq.replace("N", "")`
split into columns	`line.split("\t")`
join with tabs	`"\t".join(pieces)`
complement	`seq.translate(str.maketrans("ACGT", "TGCA"))`
print with values	`print(f"{name}\t{length}\t{gc:.2f}")`

Exercises

Write each answer as a script (`ex1.py`, ...) with a comment at the top saying what it does. Use this sequence where a sequence is needed:

```
seq = "ATGAAACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGA"
```

1. **Calculator.** A haploid yeast genome is about 12,100,000 bp. A sequencing run gives 20,000,000 reads of 150 bp. Print the coverage (total bases / genome size) with one decimal place.
2. **Types.** Predict, then check in the interpreter, the result and type of each of: `7 / 7`, `7 // 2`, `"7" * 2`, `int("7") * 2`, `float("7") / 2`, `str(7) + "7"`.
3. **Counting bases.** Print the length of `seq`, the count of each of A, C, G and T, and check that the four counts add up to the length.
4. **GC content.** Print the GC content of `seq` as a percentage with one decimal place, in the form `GC: 12.3%`.
5. **Codons.** Print the first codon, the last codon, and the 5th codon of `seq`. Is the length of `seq` a multiple of 3? Print how many complete codons it has.
6. **Reverse complement.** Print the reverse complement of `seq`. Then take the reverse complement of your result and check that you get `seq` back.
7. **Parsing a line.** Here is one line of the yeast GFF file (columns are separated by tabs):

   ```
   line = "chrI\tSGD\tgene\t335\t649\t.\t+\t.\tID=YAL069W;Name=YAL069W\n"
   ```

   Split it into columns and print the chromosome, the feature type, the strand, the gene ID (from the last column) and the length of the gene. GFF coordinates count from 1 and include both ends, so length = end - start + 1.
8. **Formatted report.** Using the values from exercise 7, print one tab-delimited line `gene  chrom  start  end  length  strand`, then the same information as a sentence: `YAL069W is on chrI (+) from 335 to 649 (315 bp).`

Hints and some solutions

- Exercise 1: `20000000 * 150 / 12100000` is about 247.9.
- Exercise 5: the last codon is `seq[-3:]`; the 5th codon is codon number 4 counting from 0, so `seq[12:15]`. `len(seq) % 3 == 0` would tell you if it is a multiple of 3 (`==` is explained in Python II); for now, print `len(seq) % 3`.
- Exercise 7:

  ```
  line = "chrI\tSGD\tgene\t335\t649\t.\t+\t.\tID=YAL069W;Name=YAL069W\n"
  cols = line.strip().split("\t")
  chrom = cols[0]
  feature = cols[2]
  ```

```
start = int(cols[3])
end = int(cols[4])
strand = cols[6]
gene_id = cols[8].split(";")[0].split("=")[1]
length = end - start + 1
print(chrom, feature, strand, gene_id, length)
```

```
chrI gene + YAL069W 315
```

Next

Python II: making decisions with `if`, lists, `for` loops, and reading and writing files - which is how we process whole data files instead of one line at a time.

Further reading

- The official Python tutorial, sections 3 ("An informal introduction") and 7.1 (formatting output)
- Ekmekci, McAnany and Mura (2016) An Introduction to Programming for Bioscientists: A Python-Based Primer. PLoS Comp Bio. 10.1371/journal.pcbi.1004867
- Rosalind - bioinformatics problems to practice on; the first few (counting nucleotides, transcribing DNA, reverse complement) use only what is in this lecture