python3 basics: dictionary tricks…

to create a dictionary explicitly, use {:}

e.g. x = {} ### empty dictionary

e.g. x = {'key0':'value0', 'key1':'value1', 'key2':'value2'}

to access a dictionary value, use a (unique) key

e.g. x['key']

to add a key/value pair to a dictionary, use =

e.g. x['key'] = y

to modify a value in a dictionary, use =

e.g. x['key'] = y

python3 basics: …dictionary tricks…

to delete an element from a dictionary, use del

e.g. del x['key0']

to test if an key is present in a dictionary, use in

e.g. y in x

to extract the keys from a dictionary, use keys()

e.g. list(x.keys())

to extract the values from a dictionary, use values()

e.g. list(x.values())

python3 basics: …dictionary tricks

to iterate a dictionary, use for

```
e.g. for y in x.keys(): ### key (y)
e.g. for y in x.values(): ### value (y)
e.g. for y, z in x.items(): ### key (y), value (z)
```

to sort dictionary keys by their corresponding values:

```
{k: v for k, v in sorted(x.items(), key=lambda item: item[1])}
```

python3 basics: math

x = y	set
x + y	add
x - y	subtract
x * y	multiply
x / y	divide (output float)
x // y	divide (output int)
x % y	modulus
x [+-*/%]= y	add/subtract/multiply/divide/modulus variable
x**y	to the power of

python3 basics: text

x = y	set
x += y	append
x + y	concatenate
x[1:3]	substring
f"text {x}"	format string
f"text {x.upper()}"	format string upper case
f"text {x.lower()}"	format string lower case
f"text {x.title()}"	format string title case
f"text {x:,}"	format number with commas
f"text {x:.6f}"	format number with fixed decimal digits

python3 basics: regular expressions

import re	import the regular expression module
x = re.compile('re')	compile a regular expression as x
re.search(x, y)	test if regex x exists in string y
re.sub(x, y, z)	replace regex x in string z with y
re.escape(y)	escape regex special characters in y

python3 basics: if/else

condition:

x==y; x!=y; x>=y; x<=y; x>y; x<y; x is y; x not y

x in y; re.search('re', x)

(); and; or

if condition: action

if condition: action; else: action

if condition: action; elif condition: action

if condition: action; elif condition: action; else: action

python3 basics: loops

while condition: action

    e.g. while x == True: action

range(x, y, z)   from x to <y counting by z

for value in variable: action

    e.g. for x in range(0, 5): action

loop keywords:

break	end the loop immediately
continue	skip to the next loop iteration

python3 basics: functions

use functions to avoid code repetition

use functions to isolate complex logic

e.g. def x(y, z): action

e.g. def x(y, z): return action

e.g. def x(y, z): action; return variable

functions can be nested

(most) variables are passed as references

changes in the function are observed everywhere

python3 basics: variable scopes

variables are global

unless declared within a function or class

local definition overrides global definition

use ‘global’ to access a global variable inside a function

use ‘nonlocal’ to modify a nested variable inside a function

python3 basics: useful keywords

def	define a function or class
float	convert to a floating point number
int	convert to an integer number
isinstance	test if variable is a particular type
len	length of a string, list, or dictionary
list	convert to a list
print	prints strings (default to STDOUT)
str	convert to a string
try	allows for recovery from a crash

python3 basics: useful modules…

from thefuzz import fuzz	approximate string matching
import datetime	datetime calculations
import decimal	precision floating point numbers
import ftfy	fixes text encoding problems
import getopt	parse command–line options
import itertools	fast looping tools
import json	JSON parsing and output
import lzma	read/write xz compression
import math	‘advanced’ math functions

python3 basics: …useful modules

import matplotlib	graph and visualization tools
import numpy	vector–based math functions
import os	(portable) operating system interfaces
import random	random numbers
import re	regular expressions
import sys	operating system interfaces
import time	timing and reporting
import textwrap	inserts newlines for nice printing

awk => python3 examples

```
awk -F'\t' '{print $3}'

python3 -c 'import
sys,re;[sys.stdout.write(line.strip().split("\t")[2]+"\n") for
line in sys.stdin]'

awk -F'\t' 'BEGIN{OFS="\t"}{print $3,$5}'

python3 -c 'import
sys,re;[sys.stdout.write("\t".join([line.strip().split("\t")[x]
for x in [2,5]])+"\n") for line in sys.stdin]'
```

…

What language should one use?

It depends…

use one that is fit to purpose

- any Turing complete language will work
- but some are better suited for particular problems

consider programming skill versus effort required

consider code longevity

consider difficulty with dependencies

efficiency has a real environmental effect

	Energy (J)
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time (ms)
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Pereira et al. (2021; https://doi.org/10.1016/j.scico.2021.102609)

how to think like a programmer

(1) determine what problem you are trying to solve

    specific problem, general solution

(2) break it down into (very) small tasks

(3) write out the steps needed to accomplish each task

    instructions for a (very simple minded and literal) person

(4) modify to match builtin functions and data structures

(5) convert steps to computer code

[(6) be persistent]

pseudocode

think about what differentiates things

- e.g. What makes x different from background text?

make a minimal model

- What does each step do?
- Why it needs to be done?
- What is the simplest way to do it? (language specific)

pseudocode: example (in python3)…

convert DNA FASTA file to its reverse complement

(1) read FASTA file

(2) make reverse complement

(3) output new FASTA file

pseudocode: …example (in python3)…

(1) read FASTA file

- (a) get file name from user
- (b) open file
- (c) read line by line
- (d) differentiate between labels and sequence
- (e) store labels and (cleaned) sequence in RAM

pseudocode: …example (in python3)…

(1) read FASTA file

(a) get file from user

```
import getopt
```

(b) open file

```
with open('input', mode = 'rt') as file:
```

(c) read line by line

```
for line in file:
```

pseudocode: …example (in python3)…

(1) read FASTA file

(d) differentiate between labels and sequence

```
re.compile('^>')
re.search()
```

(e) accumulate multiple lines of (cleaned) sequence

```
re.compile('[^ABCDGHKMNRSTVWY]')
x +=
re.sub()
```

pseudocode: …example (in python3)…

(2) make reverse complement

```
c = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A'...}
''.join(c[n] for n in reversed(x))
```

(3) output name and sequence immediately (saves memory)

```
print()
```

sequence search

DNA/RNA/protein sequences are ‘special’ text

- case is meaningless (sometimes indicates sequence/alignment quality)
- DNA orientation is not (usually) important
  - sequences are archived in arbitrary orientation
  - [RNA/protein sequences have just one orientation]
- commonly coded as letters, but could be numbers, etc.

‘query’ sequence (entire or fragments) used to find ‘reference’ sequence(s)

reference sequence annotations/metadata are (usually) the desired output

sequence search: algorithms…

exact substring matching (e.g. grep)

- one reference sequence per line, query DNA twice* (both orientations)
- will find exact reference sequence matches only
- of limited use (e.g. eDNA metabarcoding with *rbcL*)

inexact substring matching (e.g. tre-agrep)

- one reference sequence per line, query DNA twice* (both orientations)
- specify maximum allowable number of mismatches
- will find inexact reference sequence matches only
  - similar results as megaBLAST

sequence search: …algorithms…

pairwise alignment (e.g. SEQHP)

- query DNA twice* (both orientations)
- local align query to each reference sequence
- score based on query/reference alignment differences
  - uniform scoring used for each position
- rank query/reference alignments using score
  - can compute probability of match statistics

sequence search: …algorithms…

hidden Markov model (e.g. HMMER)

query DNA twice* (both orientations)

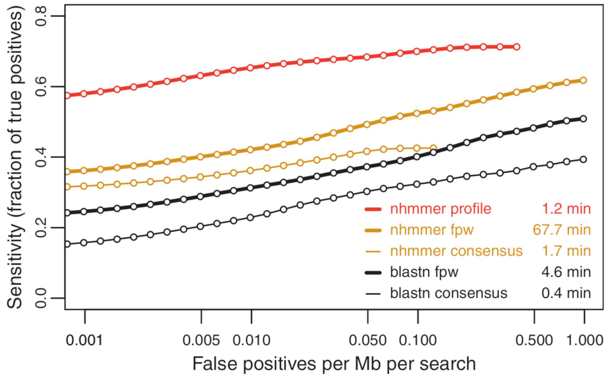
local align query to probabilistic reference models

constructed from sequence alignments

or a single sequence plus a substitution matrix

position–specific score for query/reference differences

(assumes multiple sequence alignments are possible)



(Wheeler et al. 2013; https://doi.org/10.1093/bioinformatics/btt403)

sequence search: ...algorithms...

kmers absence/presence (e.g. FACS)

- query DNA twice* (both orientations)
- count number of matching kmers
- rank query/reference match using kmer count

Method	*K*-mer size	Time (min)	Sensitivity (%)	Specificity (%)
SSAHA2/454[a]	12	32.4	98.6	98.9
BLAT/11occ[a]	11	12.5	99.8	100
BLAT/11occ/fastMap[a]	11	1.5	43.6	100
BLAT/11occ/fastMap[b]	11	1.5	66.4	100
FACS[b]	21	1.7	98.1	100
FACS[c]	21	1.7	99.8	100

(Stranneheim et al. 2010; https://doi.org/10.1093/bioinformatics/btq230)