PRACOWNIA INFORMATYCZNA



Wykład 2: Reprodukowalność analizy i narzędzia pracy

dr inż. Bartosz Czech

Katedra Genetyki · Uniwersytet Przyrodniczy we Wrocławiu

bartosz.czech@upwr.edu.pl

Plan wykładu

1. Reprodukowalność analizy
2. Skrypty powłoki
3. Kontrola wersji: Git i GitHub
4. Środowiska i pakiety: Conda
5. Konteneryzacja: Docker
6. Python: Biopython i własne narzędzia wiersza poleceń
7. Klaster obliczeniowy: HPC, SLURM i Apptainer (przegląd)
8. Pipeline'y: Snakemake
9. Wizualizacja i raporty: R oraz Quarto
10. Dobre praktyki reprodukowalności

Krótka powtórka z Wykładu 1

- Praca w terminalu: polecenia tekstowe zamiast interfejsu graficznego
- Polecenie ma budowę: `program opcje argumenty`
- Dane przetwarza się potokami — wyjście jednego programu jest wejściem następnego
- Podstawowe formaty: FASTA, FASTQ, GFF, BED, VCF, SAM/BAM
- Środowisko `pracownia` z Modułu 0: powłoka `bash`, `conda` i narzędzia kursu

1. Reprodukowalność analizy

Reprodukowalność wyniku

Reprodukowalność (*reproducibility*)

Analiza jest reprodukowalna, jeżeli inna osoba, korzystając z tych samych danych wejściowych i zapisanego opisu wykonanych kroków, otrzyma ten sam wynik.

- Reprodukowalność jest warunkiem zaufania do wyników
- Reprodukowalność zależy jednocześnie od trzech składowych: danych, kodu i środowiska
- Jeśli zabraknie którejkolwiek z nich, wynik przestaje być odtwarzalny

Źródła trudności

- Wynik zależy nie tylko od kodu, ale także od **wersji** każdego użytego narzędzia
- Dwie wersje tego samego programu (np. 1.9 i 2.0) mogą dać różne wyniki dla tych samych danych
- Analiza działająca na jednym komputerze może zawieść na innym, jeśli zainstalowano tam inne wersje bibliotek
- Bez zapisu kroków po pewnym czasie nawet autor nie odtworzy, jakie polecenia i w jakiej kolejności doprowadziły do ryciny w publikacji

Kryzys reprodukowalności

Ioannidis i in. (2009) sprawdzili 18 artykułów z *Nature Genetics* (2005–2006) z analizą ekspresji genów na mikromacierzach. Niezależne zespoły próbowały odtworzyć z każdego artykułu jedną tabelę lub rycinę, korzystając z danych i opisu metod podanych przez autorów: udało się to w 2 przypadkach, częściowo w 6, a w 10 wcale. Główne przyczyny: niedostępne dane oraz niepełny opis przetwarzania i analizy.

Przykład z praktyki

- Wynik opublikowano, ale nie można go już odtworzyć
- Drobna poprawka, np. na prośbę recenzenta, wymaga powtórzenia całej analizy
- Każdego z tych problemów można było uniknąć, zapisując kod i wersje narzędzi oraz zachowując niezmienione dane surowe

Reprodukowalność a poprawność

Ważne rozróżnienie

Reprodukowalność oznacza jedynie, że powtórzenie analizy daje ten sam wynik. Nie gwarantuje ona, że wynik jest poprawny: błędną analizę również można wiernie powtórzyć.

- Reprodukowalność jest warunkiem koniecznym, nie wystarczającym
- Jej wartość polega na tym, że jawny zapis kroków pozwala znaleźć i poprawić błąd
- Analizy niemożliwej do powtórzenia nie można też zweryfikować

Trzy filary reprodukowalności

Kod

- Zapisany w skryptach
- Pod kontrolą wersji
- Czytelny i opisany

Środowisko

- Znane wersje narzędzi
- Conda lub kontener
- Odtwarzalne na innym komputerze

Dane

- Dane surowe niezmienione
- Opisane pochodzenie
- Wyniki generowane przez kod

Narzędzia omawiane na tym wykładzie służą zapewnieniu każdego z tych filarów.

2. Skrypty powłoki

Ograniczenia pracy ręcznej

- Pojedynczy plik można przetworzyć ręcznie, wpisując kolejne polecenia w terminalu
- W praktyce próbek jest zwykle **dziesiątki albo setki**, a te same kroki trzeba wykonać dla każdej
- Ręczne powtarzanie kroków jest czasochłonne i **podatne na błędy**, takie jak pominięcie pliku czy pomyłka w opcji
- Rozwiązaniem jest jednorazowe zapisanie kroków w **skrypcie** i uruchomienie go dla wszystkich plików

Reguła praktyczna

Sekwencję poleceń wykonywaną po raz drugi warto zapisać w skrypcie.

Skrypt powłoki

Skrypt powłoki to plik tekstowy z poleceniami, które powłoka wykonuje kolejno, od pierwszego wiersza do ostatniego.

```
#!/usr/bin/env bash
# analiza.sh — pierwszy skrypt

echo "Zaczynam analizę"
ls *.fastq.gz
echo "Gotowe"
```

- Pierwszy wiersz (shebang) wskazuje program, który wykona plik; `/usr/bin/env bash` wybiera `bash` z katalogów w `PATH`, także ze środowiska Conda
- Wiersze zaczynające się od `#` to komentarze, pomijane przez powłokę
- Pozostałe wiersze powłoka wykonuje tak samo jak polecenia wpisane w terminalu

Uruchomienie skryptu

Skrypt można uruchomić przez interpreter albo jako program wykonywalny.

```
bash analiza.sh          # sposób 1: uruchomienie przez bash

chmod +x analiza.sh      # sposób 2: nadanie prawa wykonywania
./analiza.sh             # uruchomienie skryptu jako programu
```

- `chmod +x` nadaje plikowi prawo wykonywania (opisane na Wykładzie 1)
- `./` wskazuje plik w bieżącym katalogu; bez `/` powłoka szuka polecenia tylko w katalogach z `PATH`

Obsługa błędów w skrypcie

Domyślnie bash wykonuje kolejne polecenia także po błędzie. W analizie danych jest to niebezpieczne, dlatego skrypt powinien zatrzymać się przy pierwszym błędzie.

```
#!/usr/bin/env bash
set -euo pipefail

# -e  : przerwanie przy pierwszym błędzie
# -u  : błąd, gdy użyto niezdefiniowanej zmiennej
# -o pipefail : wykrywanie błędu wewnątrz potoku
```

Dobra praktyka

Dzięki tym trzem opcjom błąd przerywa skrypt z komunikatem, a niekompletny wynik nie zostaje zapisany jako poprawny.

Zmienne

- Zmienna przechowuje wartość pod nazwą, dzięki czemu wartość podaje się w jednym miejscu
- Przypisanie **bez spacji** wokół znaku równości
- Odwołanie przez $ przed nazwą

```
katalog="wyniki"
probka="RNA_01"

echo "Zapisuję do $katalog/$probka.txt"
mkdir -p "$katalog"
```

Częsty błąd

Zapis `katalog = "wyniki"` (ze spacjami) **nie jest przypisaniem**: bash potraktuje `katalog` jako nazwę polecenia. Przypisanie ma postać `nazwa=wartość`, bez spacji.

Argumenty skryptu

Skrypt nie powinien zawierać nazw plików wpisanych na stałe, lecz przyjmować je jako argumenty.

```
#!/usr/bin/env bash
# licz.sh — liczba odczytów w pliku FASTQ
set -euo pipefail
plik="$1"                       # pierwszy argument

liczba=$(zcat "$plik" | wc -l)
echo "$plik: $((liczba / 4)) odczytów"
```

- `$1`, `$2`, `$3` to kolejne argumenty z wiersza wywołania
- `$@` to lista wszystkich argumentów, `$#` — ich liczba
- Wywołanie: `bash licz.sh probka.fastq.gz`

Pętla for

```
for plik in *.fastq.gz; do
    liczba=$(zcat "$plik" | wc -l)
    echo "$plik: $((liczba / 4)) odczytów"
done
```

- `for plik in *.fastq.gz` — wykonanie ciała pętli dla każdego pasującego pliku
- W kolejnych przebiegach pętli zmienna `$plik` zawiera kolejne nazwy plików
- Ten sam kod obsługuje jeden plik i sto plików

Instrukcja warunkowa

```
plik="$1"

if [[ -f "$plik" ]]; then
    echo "Przetwarzam $plik"
else
    echo "Błąd: nie ma pliku $plik" >&2
    exit 1
fi
```

- `[[ -f "$plik" ]]` sprawdza, czy istnieje zwykły plik o tej nazwie
- `>&2` kieruje komunikat na **standardowe wyjście błędów** (*stderr*), a nie na standardowe wyjście
- `exit 1` kończy skrypt z kodem błędu; dalsze polecenia nie są wykonywane

Dobre praktyki pisania skryptów

Zalecane

- Shebang `#!/usr/bin/env bash`
- Nagłówek z opisem działania skryptu
- Nazwy plików jako argumenty
- Cudzysłowy wokół `"$zmienna"`
- `set -euo pipefail` na początku

Do unikania

- Ścieżki wpisane na stałe
- Nadpisywanie danych surowych
- Brak komentarzy
- Spacje w `nazwa = wartość`
- Brak obsługi błędów

3. Kontrola wersji: Git

Ręczne wersjonowanie plików

Katalog projektu po miesiącu pracy:

```
analiza.sh
analiza_v2.sh
analiza_final.sh
analiza_final_poprawiona.sh
analiza_naprawde_final.sh
analiza_final_DZIALA.sh
```

Problemy

- Który plik jest aktualny?
- Co zmieniło się między wersjami?
- Jak wrócić do wersji sprzed tygodnia?
- Jak połączyć zmiany wprowadzone przez współpracownika w jego kopii?

Kopie plików z dopiskami w nazwie to ręczna i zawodna forma kontroli wersji.

Czym jest kontrola wersji

Kontrola wersji (*version control*)

System kontroli wersji zapisuje historię zmian w projekcie: kto, kiedy i co zmienił. Pozwala wrócić do dowolnego wcześniejszego stanu i łączyć pracę wielu osób.

- Git jest obecnie najpowszechniej stosowanym systemem kontroli wersji; w kursie używa się go z wiersza poleceń
- Zamiast wielu kopii pliku przechowuje się jeden plik i pełną historię jego zmian
- Każdy zapis historii (commit) ma datę, autora i opis
- Można porównać dowolne wersje i przywrócić każdą z nich

Trzy obszary w Gicie

Katalog roboczy

Edytowane pliki w zwykłym katalogu na dysku.

Poczekalnia

(staging area) Zmiany wybrane do zapisania w następnym commicie.

Repozytorium

Zapisana historia: wszystkie commity projektu.

`git add` przenosi zmiany do poczekalni, `git commit` zapisuje je w historii.

Pierwsze kroki: konfiguracja, init, add, commit

```
git config --global user.name "Jan Kowalski"      # jednorazowo: autor zmian
git config --global user.email "jan.kowalski@example.org"
git config --global init.defaultBranch main      # gałąź główna: main

git init                        # utworzenie repozytorium w bieżącym katalogu
git add analiza.sh              # dodanie pliku do poczekalni
git commit -m "Pierwsza wersja skryptu analizy"
```

- `git config --global` — jednorazowo; bez danych autora `git commit` kończy się błędem
- `git init` tworzy repozytorium (ukryty katalog `.git`); `git add` wybiera zmiany do commita; `git commit -m` zapisuje je z opisem

Opis commita

Nieprecyzyjny: „Poprawki”. Precyzyjny: „Popraw liczenie odczytów dla plików skompresowanych”.

Polecenia używane na co dzień

```
git status                    # stan plików względem ostatniego commita
git diff                      # zmienione wiersze (jeszcze poza poczekalnią)

git add licz.sh
git commit -m "Dodaj obsługę argumentu z nazwą pliku"

git log --oneline             # historia commitów, po jednym w wierszu
```

- `git status` — najczęściej używane polecenie; pokazuje bieżący stan plików
- `git diff` — zmiany jeszcze niedodane do poczekalni; `git diff --staged` — zmiany w poczekalni
- `git log` — cała historia projektu

Pliki spoza repozytorium

Dane poza repozytorium

Git jest przeznaczony do plików tekstowych: skryptów, konfiguracji, notatek. Nie nadaje się do dużych plików danych (FASTQ, BAM, wyniki pośrednie), ponieważ repozytorium rosłoby wtedy do rozmiarów utrudniających pracę.

Pliki pomijane przez Gita wymienia się w pliku `.gitignore`:

```
*.fastq.gz
*.bam
wyniki/
.DS_Store
```

- Repozytorium zawiera kod, który tworzy wyniki, a nie same wyniki
- Dane przechowuje się osobno; repozytorium opisuje, jak je przetworzyć

Repozytorium zdalne: GitHub

- Serwisy **GitHub** i GitLab przechowują repozytoria Git na serwerach w sieci
- Zapewniają **kopię zapasową**, dostęp do historii z każdego komputera i możliwość współpracy
- Publiczne repozytorium jest obecnie typowym sposobem udostępniania kodu towarzyszącego publikacji

```
git clone https://github.com/uzytkownik/projekt.git   # pobranie repozytorium

git push                        # wysłanie własnych commitów na serwer
git pull                        # pobranie zmian wprowadzonych przez innych
```

Gałęzie

- **Gałąź** (*branch*) to osobna linia rozwoju projektu, na której wprowadza się zmiany bez naruszania wersji głównej
- Gotową i sprawdzoną zmianę **scala się** z gałęzią główną
- Pozwala to sprawdzić nowe podejście bez ryzyka uszkodzenia działającej analizy

```
git switch -c nowy-pomysl      # utworzenie gałęzi i przejście na nią
git switch main                # powrót na gałąź główną
git merge nowy-pomysl          # scalenie zmian z gałęzią główną
```

`git switch -c` działa tak samo jak polecenie `git checkout -b` używane w Module 6.

Przykład

Inny sposób normalizacji danych można wypróbować na osobnej gałęzi. Jeśli wyniki są lepsze, gałąź scala się z główną; jeśli gorsze, porzuca się ją, a wersja główna pozostaje bez zmian.

Zawartość repozytorium

Do repozytorium

- Skrypty i kod analiz
- Pliki konfiguracyjne
- Opis środowiska (Conda)
- Dokumentacja, README

Poza repozytorium

- Dane surowe (FASTQ, BAM)
- Duże wyniki pośrednie
- Wrażliwe dane pacjentów
- Pliki tymczasowe i systemowe

Zasada: repozytorium przechowuje **kod i opis analizy**, a nie dane ani wyniki.

4. Środowiska i pakiety: Conda

Konflikt wersji

- Projekt A potrzebuje programu w wersji 1.0, projekt B tego samego programu w wersji 2.0
- W jednej wspólnej instalacji nie mogą współistnieć obie wersje
- Narzędzia bioinformatyczne mają dziesiątki **zależności** — bibliotek, na których się opierają
- Instalacja jednego programu może zmienić wspólne zależności i uniemożliwić działanie innego

Konflikty zależności (*dependency hell*)

Ręczne dobieranie wzajemnie zgodnych wersji dziesiątek bibliotek jest w praktyce niewykonalne, a nieudokumentowane wersje często uniemożliwiają powtórzenie analizy.

Osobne środowiska

Środowisko (*environment*)

Odizolowany zestaw programów i bibliotek w konkretnych wersjach. Każdy projekt ma własne środowisko, niezależne od pozostałych.

- Projekty A i B mają osobne środowiska, więc ich wersje **nie kolidują**
- Wersje są jawne i zapisane, więc środowisko można **odtworzyć** na innym komputerze
- Menedżer środowisk używany w kursie to **Conda**

Conda w skrócie

- Conda jest jednocześnie **menedżerem pakietów** i **menedżerem środowisk**
- Instaluje nie tylko pakiety Pythona, ale też programy napisane w innych językach (R, C, Java)
- Automatycznie dobiera zgodne wersje zależności
- W kursie Condę instaluje się w dystrybucji **Miniforge** (Moduł 0, omówienie na Wykładzie 1)

Miniforge, mamba, conda

Miniforge to minimalna dystrybucja Condy, domyślnie ustawiona na kanał `conda-forge`. Zawiera też `mamba` — szybszy zamiennik polecenia `conda` o tej samej składni.

Tworzenie środowiska i kanały

```
conda config --add channels bioconda      # kanały: konfiguracja jednorazowa
conda config --add channels conda-forge   # conda-forge na pierwszym miejscu
conda config --set channel_priority strict

conda create -n rnaseq salmon samtools    # nowe środowisko z narzędziami
conda activate rnaseq                     # aktywacja środowiska
conda deactivate                          # dezaktywacja
```

- Kanał — repozytorium, z którego Conda pobiera pakiety: Bioconda (narzędzia bioinformatyczne) korzysta z conda-forge (ogólne oprogramowanie naukowe), dlatego conda-forge ma najwyższy priorytet
- Po aktywacji znak zachęty pokazuje `(rnaseq)`; pakiety instaluje się tylko w tym środowisku

Przegląd i usuwanie środowisk

```
conda env list                # lista środowisk
conda list                    # pakiety w aktywnym środowisku
conda remove -n rnaseq --all  # usunięcie całego środowiska
```

- `conda env list` — lista wszystkich środowisk
- `conda list` — spis narzędzi i ich wersji w bieżącym środowisku
- Nieużywane środowisko można usunąć i w razie potrzeby odtworzyć z pliku

Pakiety z pip w środowisku Conda

- Python ma własny menedżer pakietów — pip — instalujący z repozytorium PyPI
- Jeśli pakiet jest dostępny tylko w PyPI, w środowisku Conda instaluje się go przez `pip`
- Zasada: najpierw przez `conda` instaluje się wszystko, co jest dostępne w kanałach, a `pip` stosuje się na końcu

```
conda activate rnaseq
conda install pysam                # najpierw conda (kanały jak wyżej)
pip install jakis-pakiet-z-pypi    # pip tylko dla pakietów spoza kanałów
```

Kolejność instalacji

Instalowanie pakietów przez `pip` i `conda` w przypadkowej kolejności może doprowadzić do niespójnego środowiska. Instalacja najpierw przez `conda`, a potem przez `pip` ogranicza ryzyko konfliktów.

Reprodukowalne środowisko: environment.yml

Środowisko opisuje się w pliku tekstowym, z którego można je odtworzyć na innym komputerze. Plik zwykle pisze się ręcznie, z przypiętymi wersjami:

```
name: rnaseq
channels: [conda-forge, bioconda]
dependencies:
  - python=3.11
  - salmon=1.10.2
  - samtools=1.19
```

```
conda env create -f environment.yml    # odtworzenie środowiska z pliku
```

- `conda env export` — pełna lista pakietów z numerami buildów; odtworzenie gwarantowane tylko w tym samym systemie operacyjnym
- `conda env export --from-history` — tylko pakiety zamówione wprost; plik przenośny

Conda: dobre praktyki

Zalecane

- Jedno środowisko na projekt
- Opis w `environment.yml`
- Wersje pakietów w pliku
- Plik w repozytorium Git

Do unikania

- Instalowanie pakietów w środowisku `(base)`
- Mieszanie `pip` i `conda` bez potrzeby
- Instalacje bez zapisu w pliku środowiska
- Jedno środowisko dla wszystkich projektów

Opisane i odtwarzalne środowisko to drugi filar reprodukowalności.

5. Konteneryzacja: Docker

Ograniczenia Condy

- Conda zarządza pakietami, ale nie obejmuje **całego systemu**: bibliotek systemowych, konfiguracji ani narzędzi spoza jej kanałów
- Niektóre narzędzia wymagają konkretnego systemu operacyjnego albo trudnych do odtworzenia zależności
- Bywa potrzebna gwarancja, że analiza zadziała **tak samo** na laptopie, serwerze i klastrze

Kontener (*container*)

Odizolowane środowisko uruchomieniowe zawierające aplikację wraz z bibliotekami i plikami systemu; przy tej samej architekturze procesora działa tak samo na każdym komputerze.

Kontener a maszyna wirtualna

Maszyna wirtualna

- Wirtualizuje cały komputer
- Własny, pełny system operacyjny
- Duże zużycie zasobów, długi start

Kontener

- Współdzieli jądro z gospodarzem
- Zawiera tylko to, czego potrzebuje aplikacja
- Małe zużycie zasobów, start w kilka sekund

Kontener izoluje programy i biblioteki, lecz korzysta z jądra gospodarza: izolacja jest słabsza niż w maszynie wirtualnej, za to zużycie zasobów znacznie mniejsze.

Obraz i kontener

- Obraz (*image*) to niezmienny szablon zawierający aplikację i wszystkie jej zależności
- Kontener to uruchomiona instancja obrazu
- Z jednego obrazu można uruchomić wiele identycznych kontenerów
- Obrazy pobiera się z rejestrów, na przykład Docker Hub albo quay.io (w tym obrazy BioContainers)

Docker w praktyce

```
docker pull quay.io/biocontainers/samtools:1.19--h50ea8bc_0

docker run --rm quay.io/biocontainers/samtools:1.19--h50ea8bc_0 \
    samtools --version
```

- `docker pull` pobiera gotowy obraz z rejestru; `docker run` uruchamia w nim polecenie, a `--rm` usuwa kontener po zakończeniu
- Projekt BioContainers udostępnia tysiące narzędzi bioinformatycznych jako gotowe obrazy w rejestrze `quay.io/biocontainers`
- Tag (`wersja--identyfikator_builda`) należy odczytać z rejestru

Dockerfile

Obraz można zbudować samodzielnie z pliku tekstowego zwanego `Dockerfile`.

```
FROM ubuntu:24.04
# wersja samtools: ta z repozytorium Ubuntu 24.04
RUN apt-get update && apt-get install -y samtools
COPY analiza.sh /opt/analiza.sh
CMD ["bash", "/opt/analiza.sh"]
```

- `FROM` — system bazowy; `RUN` — polecenia budujące obraz (instalacje)
- `COPY` — kopiowanie pliku z komputera do obrazu; `CMD` — domyślne polecenie przy starcie kontenera
- Dockerfile jest plikiem tekstowym, więc przechowuje się go w repozytorium Git obok kodu

Conda a kontener

Conda wystarczy, gdy

- Narzędzia są dostępne w kanale Bioconda
- Praca odbywa się lokalnie
- Istotna jest prosta konfiguracja

Kontener, gdy

- Potrzebna jest izolacja całego środowiska systemowego
- Analiza ma działać tak samo na różnych komputerach
- Zależności są trudne do odtworzenia

Oba rozwiązania służą temu samemu celowi: **odtwarzalności** środowiska.

6. Python: Biopython i własne narzędzia wiersza poleceń

Ograniczenia basha

- Bash dobrze nadaje się do **łączenia narzędzi** i przetwarzania plików w potokach
- Słabiej sprawdza się w obliczeniach, złożonej **logice** i pracy ze strukturami danych
- Do obliczeń, przekształcania danych czy wykresów potrzebny jest język programowania
- W bioinformatyce dominują dwa języki: **Python** i **R**

Podział ról

Bash łączy narzędzia i pliki, a Python i R służą do obliczeń, analizy i wizualizacji.

Dlaczego Python

- Czytelna składnia, łatwa do opanowania na początku
- Bogaty zestaw bibliotek naukowych
- Jeden z głównych języków analizy danych, uczenia maszynowego i przetwarzania sekwencji
- Python 3.11 jest w środowisku `pracownia` z Modułu 0 (`conda activate pracownia`)

Biblioteki

W kursie: biblioteka standardowa (wbudowana w Pythona) i Biopython — sekwencje i formaty biologiczne. Poza kursem warto znać z nazwy: pandas (tabele), NumPy (obliczenia), Matplotlib (wykresy).

Uruchamianie Pythona

```
python3                          # tryb interaktywny (szybkie sprawdzanie kodu)

python3 analiza.py               # uruchomienie skryptu
```

Skrypt `analiza.py` to zwykły plik tekstowy:

```
# analiza.py
liczby = [10, 20, 30]
srednia = sum(liczby) / len(liczby)
print("Średnia:", srednia)
```

- Tryb interaktywny nadaje się do szybkiego sprawdzenia pomysłu
- Do powtarzalnej analizy kod zapisuje się w skrypcie — jak w bashu

Podstawowe struktury danych

Python oferuje struktury danych, których brakuje w bashu.

```
# lista — uporządkowany ciąg wartości
geny = ["gen1", "gen2", "gen3"]

# słownik — pary klucz-wartość
ekspresja = {"gen1": 12.5, "gen2": 0.3}

print(ekspresja["gen1"])      # 12.5
```

- **Lista** przechowuje wartości w kolejności
- **Słownik** wiąże klucz z wartością — na przykład nazwę genu z poziomem ekspresji
- Na tych dwóch strukturach opiera się większość prostych analiz

Czytanie pliku i pętla

```
# liczba wierszy zaczynających się od ">" w pliku FASTA
liczba = 0
with open("sekwencje.fasta") as f:
    for linia in f:
        if linia.startswith(">"):
            liczba += 1
print("Liczba sekwencji:", liczba)
```

- `for linia in f` — odczyt pliku wiersz po wierszu, bez wczytywania całości do pamięci
- Ten sam wynik daje w bashu `grep -c`; Python ułatwia dodanie bardziej złożonej logiki

Własne narzędzie wiersza poleceń

Skrypt staje się narzędziem, gdy przyjmuje argumenty, zgłasza błędy kodem wyjścia i ma testy (Moduł 8).

```
import argparse
p = argparse.ArgumentParser(description="Udział G+C w sekwencji.")
p.add_argument("sekwencja", help="sekwencja DNA")
seq = p.parse_args().sekwencja.upper()
print(f"{100 * sum(z in 'GC' for z in seq) / len(seq):.1f}")
```

```
$ python3 procent_gc.py ATGCGC
66.7
$ python3 procent_gc.py; echo $?
usage: procent_gc.py [-h] sekwencja
procent_gc.py: error: the following arguments are required: sekwencja
2
```

- `argparse` tworzy pomoc (`-h`) i komunikat o błędzie; kod wyjścia `2` informuje powłokę i pipeline o niepowodzeniu; w Module 8 także `subprocess` i testy `unittest`

Biopython

Biopython to biblioteka do pracy z danymi biologicznymi, obsługująca typowe formaty i operacje.

```
from Bio import SeqIO

for rekord in SeqIO.parse("sekwencje.fasta", "fasta"):
    print(rekord.id, len(rekord.seq))
```

- `SeqIO.parse` czyta FASTA, FASTQ, GenBank i inne formaty
- Każdy `rekord` ma identyfikator, opis i samą sekwencję
- Biblioteka parsuje formaty, więc nie trzeba pisać własnego parsera

Funkcje biblioteki

Gotowe, przetestowane funkcje: sekwencja odwrotnie komplementarna, translacja, odczyt anotacji, konwersja między formatami.

Bash, Python i R

Typowe zastosowania w bioinformatyce:

Bash (Moduły 1–4)

- Łączenie gotowych programów
- Przetwarzanie plików w potokach
- Automatyzacja powtarzalnych kroków
- Proste warunki i pętle

Python i R (Moduły 7–8 i 10)

- Obliczenia i analiza statystyczna
- Złożona logika i struktury danych
- Parsowanie sekwencji i formatów
- Uczenie maszynowe
- Ryciny publikacyjne i raporty
- Własne narzędzia i pakiety

Między Pythonem a R wybiera się zwykle według bibliotek potrzebnych w zadaniu. W kursie Python służy do parsowania sekwencji i budowy narzędzi, a R — do rycin i raportu.

7. Klaster obliczeniowy: HPC, SLURM i Apptainer (przegląd)

Ograniczenia komputera osobistego

- Dopasowanie (mapowanie) odczytów całego eksperymentu do genomu może trwać **kilka dni** na jednym komputerze
- Niektóre analizy wymagają **setek gigabajtów** pamięci operacyjnej
- Sto próbek warto przetwarzać **równolegle**, a nie kolejno
- Rozwiązaniem jest klaster obliczeniowy (HPC, *high-performance computing*)

Klaster obliczeniowy (*computing cluster*)

Zbiór połączonych komputerów (węzłów) ze wspólną przestrzenią dyskową, między które system kolejkowy rozdziela zadania użytkowników.

Sekcja poglądowa — w kursie nie korzysta się z klastra. Na zaliczeniu pisemnym obowiązują z niej tylko pojęcia: węzeł dostępowy i obliczeniowy, system kolejkowy, zadanie wsadowe (`sbatch`), podgląd kolejki (`squeue`).

Budowa klastra

Węzeł dostępowy

(login node) Służy do logowania, edycji plików i zlecania zadań; nie uruchamia się na nim obliczeń.

Węzły obliczeniowe

(compute nodes) Wykonują obliczenia; każdy ma dziesiątki rdzeni i dużą pamięć operacyjną.

- System kolejkowy (*scheduler*) rozdziela zadania między węzły i zapewnia użytkownikom równy dostęp
- Wspólny dysk jest dostępny ze wszystkich węzłów

Logowanie zdalne: SSH

Z klastrem łączy się przez SSH (*Secure Shell*) — szyfrowane połączenie tekstowe.

```
ssh login@klaster.uczelnia.pl      # logowanie na węzeł dostępowy
```

- Po zalogowaniu terminal klastra działa tak samo jak terminal lokalny
- Praca odbywa się wyłącznie w trybie tekstowym, dlatego znajomość wiersza poleceń jest niezbędna
- Pliki przesyła się poleceniami `scp` albo `rsync`

Przenoszenie danych na klaster

Przed analizą dane kopiuje się na wspólny dysk klastra.

```
scp probka.fastq.gz login@klaster:/scratch/projekt/    # pojedynczy plik

rsync -av dane/ login@klaster:/scratch/projekt/dane/   # cały katalog
```

- `scp` kopiuje pliki przez szyfrowane połączenie SSH
- Po przerwanym transferze `rsync` przesyła tylko brakujące lub zmienione pliki
- Wiele małych plików warto najpierw połączyć w archiwum (`tar`), aby przesłać jeden plik zamiast tysięcy

System kolejkowy SLURM

- Zasoby klastra są ograniczone, dlatego zadania **oczekują w kolejce**
- **SLURM** — powszechnie stosowany system kolejkowy na klastrach naukowych
- Zadanie zgłasza się wraz z wymaganymi zasobami; SLURM uruchamia je, gdy zasoby są dostępne
- Czas oczekiwania zależy od obciążenia klastra

Bez obliczeń na węźle dostępowym

Uruchomienie wymagającej obliczeniowo analizy wprost na węźle dostępowym spowalnia pracę wszystkich zalogowanych użytkowników. Obliczenia zawsze zleca się przez SLURM na węzły obliczeniowe.

Trzy sposoby uruchomienia zadania

```
srun --cpus-per-task=4 samtools sort -@ 4 -o posortowany.bam plik.bam

salloc --cpus-per-task=8 --mem=32G     # sesja interaktywna

sbatch zadanie.sh                      # zadanie wsadowe w tle
```

- `srun` — uruchomienie polecenia na węźle obliczeniowym; `-@ 4` — dodatkowe wątki `samtools`, które wykorzystują przydzielone rdzenie; `-o` — zapis wyniku do pliku
- `salloc` — rezerwacja zasobów na sesję interaktywną
- `sbatch` — zgłoszenie skryptu do wykonania w tle (najczęściej stosowane)

Skrypt wsadowy dla SLURM

```
#!/bin/bash
#SBATCH --job-name=kwantyfikacja
#SBATCH --cpus-per-task=8
#SBATCH --mem=32G
#SBATCH --time=04:00:00

source "$(conda info --base)/etc/profile.d/conda.sh"
conda activate rnaseq
salmon quant -i indeks -l A -r probka.fastq.gz -p 8 -o wynik
```

- Wiersze `#SBATCH` określają wymagane zasoby: rdzenie, pamięć i maksymalny czas
- Skrypt wsadowy nie jest sesją interaktywną, więc `conda activate` działa w nim dopiero po wierszu `source …conda.sh`; bez niego skrypt kończy się błędem
- Zgłoszenie: `sbatch zadanie.sh`

Gotowe oprogramowanie na klastrze

Klastry udostępniają część narzędzi centralnie, przez system **modułów**.

```
module avail                    # dostępne moduły
module load samtools/1.19       # załadowanie konkretnej wersji
module list                     # moduły załadowane w tej sesji
```

- `module load` udostępnia narzędzie w bieżącej sesji
- Wersja jest jawna, więc zapis pozostaje reprodukowalny
- W praktyce moduły klastra łączy się z własnymi środowiskami Conda

Apptainer

Docker w systemach współdzielonych

Docker zwykle wymaga uprawnień administratora, których użytkownik klastra obliczeniowego nie ma.
Z tego powodu w ośrodkach HPC używa się Apptainera (dawniej Singularity).

- Apptainer uruchamia kontenery z uprawnieniami zwykłego użytkownika, co jest bezpieczne w systemie współdzielonym
- Uruchamia obrazy Dockera po ich konwersji do formatu SIF
- Jest powszechnie stosowany w ośrodkach obliczeniowych

Monitorowanie zadań

```
squeue --me                # własne zadania: oczekujące i wykonywane
scancel 123456             # anulowanie zadania o podanym numerze
sacct -j 123456            # informacje o zakończonym zadaniu
```

- `squeue` — zadania oczekujące i wykonywane
- `scancel` — przerwanie zadania
- `sacct` — czas trwania, zużycie pamięci, kod zakończenia

Reguła praktyczna

Należy zamawiać tyle zasobów, ile zadanie faktycznie potrzebuje: zawyżone czas i pamięć wydłużają oczekiwanie w kolejce i ograniczają dostęp innym użytkownikom.

8. Pipeline'y: Snakemake

Analiza jako ciąg kroków

Typowa analiza RNA-seq to ciąg zależnych od siebie kroków:

1. Kontrola jakości surowych odczytów
2. Przycięcie odczytów słabej jakości
3. Dopasowanie (mapowanie) do genomu albo kwantyfikacja
4. Zliczenie ekspresji
5. Analiza różnicowa i wykresy

- Każdy krok korzysta z wyniku poprzedniego i tworzy dane wejściowe dla następnego
- Całość powtarza się dla **kilkudziesięciu próbek**

Ograniczenia skryptu bash

- Gdy piąty krok zakończy się błędem, skrypt bash trzeba uruchomić **od nowa** albo ręcznie pomijać wykonane etapy
- Zmiana jednego parametru wymaga ponownego wykonania wszystkich kroków, także niezmienionych
- **Równoległe** wykonanie niezależnych kroków wymaga dodatkowego kodu
- Przy dziesiątkach plików łatwo stracić rozeznanie, które wyniki już istnieją

Źródło problemu

Skrypt bash wykonuje polecenia kolejno, ale **nie zna zależności** między nimi (np. że wynik kroku 3 zależy od kroku 2), więc nie może ustalić, które kroki trzeba wykonać ponownie.

Menedżer pipeline'ów

Menedżer pipeline'ów (*workflow manager*)

Program, w którym analizę opisuje się jako zbiór reguł określających wejście, wyjście i sposób przekształcenia jednego w drugie; sam ustala kolejność kroków i wykonuje tylko te, które są potrzebne.

- Najpopularniejsze w bioinformatyce to Snakemake (składnia oparta na Pythonie) i Nextflow
- Zamiast kolejności kroków opisuje się cel i zależności między plikami
- Menedżer sam ustala, które kroki są potrzebne do osiągnięcia celu

Reguła w Snakemake

Reguła określa, jakim poleceniem z danego wejścia powstaje dane wyjście.

```
rule kwantyfikacja:
    input:  "dane/{probka}.fastq.gz"
    output: "wyniki/{probka}/quant.sf"
    shell:
        "salmon quant -i {config[indeks]} -l A -r {input} "
        "-o wyniki/{wildcards.probka}"
```

- `input` i `output` to deklaracja zależności między plikami
- `{probka}` to symbol wieloznaczny (*wildcard*): ta sama reguła obsługuje każdą próbkę
- `shell` to zwykłe polecenie powłoki; `{config[indeks]}` — ścieżka z pliku konfiguracyjnego (omówionego dalej)

Graf zależności

- Z reguł Snakemake buduje graf: który plik zależy od którego
- Aby utworzyć wynik końcowy, przechodzi graf od celu wstecz i sprawdza, których plików brakuje
- Uruchamia tylko te kroki, których wyniki nie istnieją albo są nieaktualne
- Kroki niezależne od siebie może wykonywać równolegle

Konsekwencja praktyczna

Po zmianie parametru ostatniego kroku i ponownym uruchomieniu Snakemake przelicza tylko ten krok, a nie całą analizę od surowych odczytów. Oszczędza to godziny obliczeń.

Plik konfiguracyjny

- Ścieżki, nazwy próbek i parametry przechowuje się w osobnym pliku konfiguracyjnym, a nie w kodzie pipeline'u
- Nowy zestaw danych wymaga zmiany konfiguracji, a nie reguł

```
# config.yaml
indeks: "referencje/indeks_salmon"
probki: ["RNA_01", "RNA_02", "RNA_03"]
```

```
# początek pliku Snakefile
configfile: "config.yaml"

rule all:
    input:
        expand("wyniki/{probka}/quant.sf", probka=config["probki"])
```

- `rule all` (pierwsza w pliku) zbiera wyniki końcowe; `expand()` tworzy ich listę dla wszystkich próbek

Uruchomienie lokalne i na klastrze

```
snakemake --cores 4                        # lokalnie, do 4 zadań jednocześnie

snakemake --executor slurm --default-resources --jobs 20   # na klastrze
```

- Ten sam opis pipeline'u działa na laptopie i na klastrze SLURM
- Na klastrze Snakemake sam zgłasza poszczególne kroki jako zadania
- Pipeline łączy narzędzia z tego wykładu: środowiska Conda, kontenery i system kolejkowy SLURM

Pipeline a pozostałe narzędzia

Korzyści

- Powtarzalność całej analizy
- Wznowienie po awarii
- Równoległe wykonanie bez dodatkowego kodu
- Jeden opis zamiast wielu skryptów

Powiązania z innymi narzędziami

- Kod pipeline'u w repozytorium Git
- Środowiska Conda
- Kroki w bashu, Pythonie lub R
- Klaster przy dużych obliczeniach

W pipeline'ie wszystkie omówione narzędzia działają razem.

9. Wizualizacja i raporty: R oraz Quarto

- R zaprojektowano do **analizy danych i obliczeń statystycznych**
- **Bioconductor** to zbiór pakietów R przeznaczonych dla bioinformatyki
- Do wizualizacji danych służy pakiet **ggplot2**, powszechnie używany do wykresów publikacyjnych

Bioconductor

Pakiety do analizy różnicowej ekspresji (DESeq2, edgeR), danych omicznych i adnotacji genomów; dobrze udokumentowane i szeroko stosowane w genomice.

> Podstaw języka R uczy przedmiot poświęcony pakietom statystycznym; Moduł 10 dotyczy rycin w `ggplot2` i raportu Quarto.

Przykład kodu w R

```
# wczytanie tabeli i średnia z kolumny
dane <- read.csv("ekspresja.csv")
mean(dane$gen1)

# prosty wykres z ggplot2
library(ggplot2)
ggplot(dane, aes(x = grupa, y = gen1)) + geom_boxplot()
```

- `read.csv` wczytuje tabelę do ramki danych
- `dane$gen1` — odwołanie do kolumny `gen1`
- ggplot2 tworzy wykres warstwami: dane, odwzorowanie estetyk, geometria

Raport a kod analizy

- Ręczne przenoszenie wyników do dokumentu jest pracochłonne i zawodne
- Dokument i kod istnieją osobno, więc łatwo o niezgodność między nimi
- Po kilku miesiącach trudno ustalić, z której wersji danych pochodzi dana rycina

Programowanie literackie (*literate programming*)

Łączenie w jednym dokumencie tekstu, kodu i wyników jego wykonania; wyniki i ryciny powstają automatycznie przy generowaniu dokumentu.

- Gdy zmieniają się dane, wystarczy wygenerować dokument ponownie, a wszystkie liczby i wykresy zostaną zaktualizowane
- Liczby w tekście pochodzą z kodu, więc są zgodne z wynikami
- Narzędzie używane w kursie: Quarto

Quarto

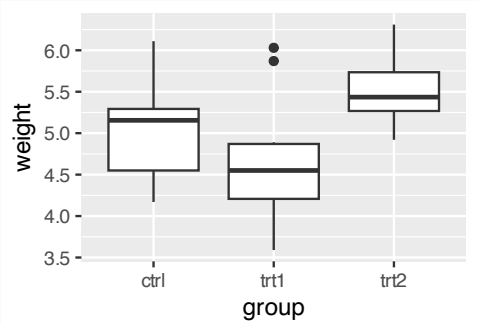
Plik źródłowy `raport.qmd`:

````
Masa roślin w grupach:

```{r}
library(ggplot2)
ggplot(PlantGrowth,
       aes(group, weight)) +
  geom_boxplot()
```
````

Wynik w wygenerowanym dokumencie:

Masa roślin w grupach:



- Przy generowaniu Quarto wykonuje kod i wstawia jego **wynik** w miejsce bloku
- Liczby w tekście również mogą pochodzić z kodu, zamiast być wpisywane ręcznie
- Z jednego źródła powstaje PDF, strona HTML albo prezentacja

Reprodukowalny raport

- Plik raportu Quarto jest **tekstowy**, więc przechowuje się go w repozytorium Git obok kodu analizy
- Inna osoba, generując go na własnym komputerze w tym samym środowisku Conda, otrzymuje **te same liczby**
- Po każdej zmianie danych lub kodu raport generuje się od nowa

Trzy filary reprodukowalności

Reprodukowalny raport łączy trzy filary z początku wykładu: dane surowe, jawny kod przetwarzania i zapisane środowisko.

10. Dobre praktyki reprodukowalności

Struktura katalogów projektu

Uporządkowany katalog projektu sam w sobie jest formą dokumentacji.

```
projekt/
  dane/
  skrypty/
  wyniki/
  environment.yml
  README.md
```

- `dane` — dane surowe, tylko do odczytu
- `skrypty` — cały kod analizy
- `wyniki` — pliki generowane przez kod
- `environment.yml` i `README.md` opisują środowisko i projekt

Układ jak na Wykładzie 1 i w pipeline'ie z Modułu 9, uzupełniony o opis środowiska.

Nienaruszalność danych surowych

Dane surowe tylko do odczytu

Dane surowe traktuje się jako niezmienne: każdy krok przetwarzania zapisuje wynik w nowym pliku. Usuniętych lub nadpisanych danych surowych zwykle nie można odzyskać.

- Katalog z danymi surowymi można zabezpieczyć przed zapisem (`chmod -R a-w dane/`)
- Po nieudanym kroku można wrócić do niezmienionych danych wejściowych
- Kopia zapasowa danych surowych jest ważniejsza niż kopia wyników: wyniki można odtworzyć z kodu, danych surowych — nie

Wyniki generowane przez kod

- Każdy wykres i każda tabela w publikacji powinny **powstawać z kodu**, który można uruchomić ponownie
- Ręczne poprawki w wynikach (edycja liczby, retusz ryciny w programie graficznym) **nie są reprodukowalne**
- Zmiana wyniku oznacza zmianę kodu i ponowne wygenerowanie wyniku

Test reprodukowalności

Czy po usunięciu katalogu `wyniki` można odtworzyć całą jego zawartość jednym poleceniem? Jeśli nie, analiza zawiera ręczny krok.

Plik README

Zawartość pliku README

- Cel projektu i pochodzenie danych
- Sposób odtworzenia środowiska (`conda env create -f environment.yml`)
- Kolejne kroki uruchomienia analizy
- Opis katalogów i plików

- Autor, który wraca do projektu po kilku miesiącach, jest w położeniu osoby z zewnątrz
- Kilka zdań opisu oszczędza wiele godzin odtwarzania wcześniejszych decyzji

Zasady FAIR

Poza reprodukowalnością analizy ważny jest sposób opisu i udostępniania danych, ujęty w zasadach **FAIR**.

- **Findable** (możliwe do znalezienia) — trwały identyfikator (numer dostępu, DOI) i opis
- **Accessible** (dostępne) — określony sposób pobrania; dostęp może wymagać zgody i uwierzytelnienia
- **Interoperable** (interoperacyjne) — standardowe formaty obsługiwane przez narzędzia
- **Reusable** (możliwe do ponownego wykorzystania) — opis, licencja, pochodzenie

Znaczenie w praktyce

Do zasad FAIR odwołują się wymagania czasopism i instytucji finansujących badania (plany zarządzania danymi); publiczne repozytoria danych, np. sekwencji, ułatwiają ich spełnienie.

Typowe pułapki

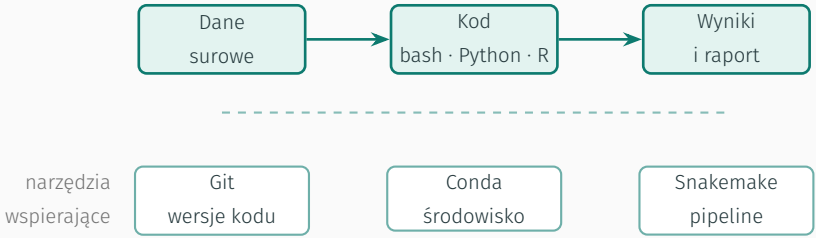
- Ręczne kroki — jeśli choć jeden krok wykonano poza kodem, analiza nie jest w pełni odtwarzalna
- Brak wersji narzędzi — „najnowsza wersja” za rok będzie inną wersją programu
- Dane w repozytorium — repozytorium nadmiernie się rozrasta
- Nadpisane dane surowe — analizy nie można powtórzyć od początku
- Kod tylko na jednym laptopie — awaria dysku oznacza utratę kodu

Wspólne rozwiązanie

Wszystkim tym pułapkom zapobiega jawny zapis wykonanych kroków: w kodzie, w opisie środowiska i w systemie kontroli wersji.

Podsumowanie

Przepływ analizy i narzędzia



Górny rząd: przepływ analizy; dolny: narzędzia, które zapewniają jej powtarzalność. Ten sam pipeline można uruchomić lokalnie albo (poza kursem) na klastrze.

Podsumowanie wykładu

- Reprodukowalność: dane, kod, środowisko
- Skrypty bash: pętle, argumenty, automatyzacja
- Git: historia zmian i współpraca
- Conda: izolowane, odtwarzalne środowiska
- Kontenery: jednakowe działanie na różnych komputerach
- Python: Biopython i własne narzędzia
- Klaster i SLURM: zasoby do dużych analiz (przegląd)
- Snakemake: pipeline z zależnościami
- R i Quarto: wykresy i reprodukowalny raport
- Dobre praktyki: struktura projektu, dane surowe, README, FAIR

Omówione narzędzia tworzą spójny zestaw służący reprodukowalności analizy.

Kolejność opanowywania narzędzi

- Narzędzia opanowuje się **stopniowo**; każde opiera się na poprzednich
- Podstawą są treści Wykładu 1: terminal, pliki, potoki
- Na nich opierają się kolejno skrypty, kontrola wersji i środowiska, Python, pipeline'y i raporty
- Każdy moduł w Moodle wprowadza kolejne narzędzia i utrwala je zadaniami

Najważniejsza zasada

Każdy krok analizy należy **zapisać** tak, aby można go było powtórzyć; omówione narzędzia służą realizacji tej zasady.

Zestawienie narzędzi z wykładu

Kod i wersje

- `bash` — skrypty i automatyzacja
- `git` — kontrola wersji
- `python`, `R` — analiza

Środowisko

- `conda` — pakiety i środowiska
- `docker`, `apptainer` — kontenery

Skala i automatyzacja

- `ssh` — praca zdalna
- `sbatch`, `squeue` — SLURM
- `snakemake` — pipeline'y

Raporty

- `quarto` — raporty łączące kod, tekst i wyniki

Szczegółowe omówienie narzędzi zawierają moduły w Moodle; klaster i SLURM omówiono tylko w tym przeglądzie i teoretycznie w Module 9.

Często zadawane pytania

- **Czy trzeba znać wszystkie te narzędzia od razu?** Nie; moduły wprowadzają je kolejno
- **Bash, Python czy R?** Bash służy do łączenia programów, a Python i R — do analizy; między Pythonem a R wybiera się według potrzebnych bibliotek
- **Po co Git przy pracy w pojedynkę?** Historia zmian pozwala wrócić do wcześniejszych wersji, a repozytorium zdalne jest kopią zapasową kodu
- **Czy w kursie jest dostęp do klastra?** Nie — wszystkie zadania wykonuje się na własnym komputerze ze środowiskiem z Modułu 0

> Narzędzi jest wiele, dlatego kurs wprowadza je kolejno, w zadaniach praktycznych.

Co dalej

- Dalsza praca odbywa się w **modułach** w Moodle, każdy z listą zadań
- Moduły wymagają samodzielnego **pisania** poleceń i skryptów
- Zaliczenie sprawdza umiejętności praktyczne, dlatego ważna jest regularna, samodzielna praca
- W razie trudności: forum kursu oraz konsultacje (termin i miejsce — Wykład 1 i Moodle)

Wykłady wprowadzające przedstawiły przegląd kursu; umiejętności kształtuje się w modułach, realizowanych kolejno.

Dziękuję za uwagę

dr inż. Bartosz Czech · bartosz.czech@upwr.edu.pl