bio334_day1_part1

May 14, 2025

1 Bio334 Practical Bioinformatics

The 2nd module, 14-16, May, 2025

1.1 Masaomi Hatakeyama

- GitHab https://github.com/masaomi/bio334_2025
- TAs: Narjes Yousefi, Kenji Yip Tong

2 Related courses

- Bio373 Next Generation Sequencing for Evolutionary Functional Genomics
- Bio609 Introduction to UNIX/Linux and Bash Scripting
- Bio610 Next-Generation Sequencing for Model and Non-Model Species

3 FGCZ courses

https://fgcz.ch/education.html

- Bio680 DNA Next Generation Sequencing
- Bio675 Transcriptomics Courses, includes single cell transcriptomics
- Bioinformatics Training, Integrative-Omics Courses, etc.

```
[ ]: import IPython.display
     IPython.display.Audio("voice/introduction.mp3")
```

Hello, my name is Masaomi Hatakeyama. This is BIO334: Practical Bioinformatics, the second module of the course. I am the lecturer for this module.

You can access all the necessary data from the GitHub repository:

https://github.com/masaomi/bio334_2025

You may not be familiar with GitHub or Git commands, but please just follow the instructions. If you have any questions, feel free to post them in the Slack channel.

Now, let's get started. Please proceed to the next cell.

4 How to download from GitHub

- Start "Terminal"
- Type the following command

```
$ git clone https://github.com/masaomi/bio334_2025
```

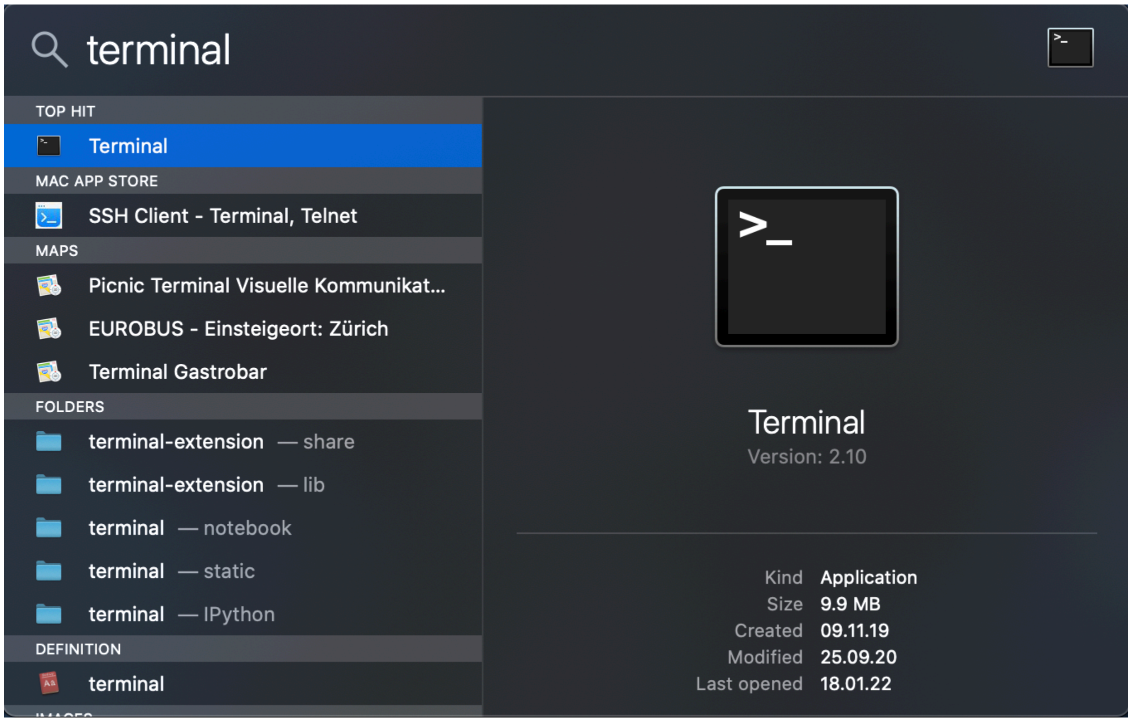
explain each folder

5 GitHub/GitLab

<img src="https://raw.githubusercontent.com/masaomi/bio334_2025/main/jupyter_notebooks/png/wha

<img src="https://raw.githubusercontent.com/masaomi/bio334_2025/main/jupyter_notebooks/png/wha

By David Whitaker, https://www.coursereport.com/blog/what-is-github



Start Jupyter Lab/Jupyter Notebook

```
$ git clone https://github.com/masaomi/bio334_2025
$ cd bio334_2025
$ jupyter lab
or
$ jupyter notebook
```

6 Copy Jupyter notebooks

- Start terminal

```
$ git clone https://github.com/masaomi/bio334_2025
```

- Update Jupyter notebooks (in the local repository)

```
$ git pull
```

```
[ ]: IPython.display.Audio("voice/git_clone.mp3")
```

After you start the Jupyter environment, you can see the “Terminal” icon on the right side of the window. Clicking the Terminal icon, start the Terminal window.

Then, type the following command:

`git clone https://github.com/masaomi/bio334_2025`

After you download all the data, you can see the new folder in the left side of the window.

7 How to use Jupyter notebook

- You can run this *cell* by typing CTRL+ENTER or SHIFT+ENTER.
  - There are two types of cell: **markdown**, **python code**
- Double click goes into the edit mode
- Down arrow key moves onto the next cell

- *Restart* when stuck
- You can edit and save your notes, and take it back to home
- If you install Jupyter in your PC, .ipynb file can work as well
- I do not recommend using Jupyter notebook/lab when you run a heavy Python calculation

```
[ ]: IPython.display.Audio("voice/jupyter_notebook.mp3")
```

In the downloaded GitHab folder, you will see a lot of Jupyter notebook files.

Just let me quickly introduce how to use the Jupyter notebook here.

In the Jupyter notebook, there are two types of cells: one is markdown cell, and the other one is Python code.

Double-clicking or hit the Enter key, goes into the edit mode. You can edit any markdown documents and Python code as you like. **Ctrl+Enter** (or **Shift+Enter**) executes the Python code.

You can see a sample Python code in the next cell. Selecting the next cell, and **Ctrl + Enter** executes the python code. Then, you will get the result under the cell. In the edit mode, **Ctrl+Enter** key goes back to the view mode again.

```
[ ]: print("hello, world")
```

You can run the cells regardless of order, but **variables** are saved.

```
[ ]: x = 123
```

```
[ ]: print(x)
[ ]: import math
     import matplotlib.pyplot as plt

     x = [i * 0.1 for i in range(63)]
     y = [math.sin(i) for i in x]

     plt.plot(x, y)
     plt.show()
```

7.0.1 Note

You can use also Google Colab (if you have a google account) * https://colab.research.google.com/

8 Goal

Implementation of

1. **Nucleotide Diversity**
2. **Segregating sites** (detecting/calling SNPs)
3. **Tajima's D**

in Python (without using any third party libraries)

9 Underlying Objective

Learning **how to construct an algorithm** through the implementation of

1. Nucleotide Diversity
2. Segregating sites (detecting/calling SNPs)
3. Tajima's D

as an example in Python

10 Option

Learning **how to use AI support tools** for programming through the implementation of

1. Nucleotide Diversity
2. Segregating sites (detecting/calling SNPs)
3. Tajima's D

as an example in Python

```
[ ]: IPython.display.Audio("voice/goal.mp3")
```

The goal of this module is the implementation of the nucleotide diversity and Tajima's D in Python.

But more importantly, you should learn how to construct an algorithm through the implementation of the nucleotide diversity and Tajima's D as an example.

I guess you have learned the basic techniques and concepts of Python in the first module. For example, what is the variable, what is the operator, method, if-statement, and so on.

The next step in learning a computer programming should be to learn how to construct an algorithm.

Now you know mathematically how to calculate the nucleotide diversity. But how do you implement it in Python?

In this module, you can learn how to construct an algorithm, by building up small blocks of code, and at the end, you will see how to solve the nucleotide diversity and Tajima's D in Python.

11 Plan

1. Day1 (through Python review): Nucleotide diversity
2. Day2 (through population genetics review): Tajima's D
3. Day3: Advanced exercise (if we have time)

12 Lecture style

in each topic 1. Example (Jupyter notebook) 2. Exercise (by yourself) 3. Advanced exercise (option, for advanced participants)

- Q&A (As needed)
- Answer code will be added in GitHub after 5pm

Key concept - *Step by step, break a problem into as small of pieces as possible* - *Divide and conquer* - *Eile mit Weile (Make haste slowly)*

```
[ ]: IPython.display.Audio("voice/lecture_style.mp3")
```

In this lecture, I divided the topics into smaller parts, like part1, part2, ..., in order to focus on each topic more simply.

In each part, I will give you an example on the Jupyter notebook first, and then you can try some exercises in the Jupyter environment by yourself at your own pace.

I just assign roughly one hour in each part, but you do not have to rush. Just see how the small code blocks are built up and connected to each other, and think why and how it calculates until you understand it.

If you have a question, please put it in a Slack channel. I will answer your question as soon as possible. The answer code for the exercises will be added in the GitHub at the end of the day.

13 Important skills (*in programming*)

behind (beyond?) the knowledge of programming language (grammar)

1. **Deduction**: Using logic (e.g., if a $\rightarrow$ b and b $\rightarrow$ c then a $\rightarrow$ c)
2. **Induction**: Generalization from specific instances (e.g., identifying patterns)

3. **Abduction**: Hypothetical reasoning (e.g., forming hypotheses based on incomplete information, using deduction and induction, the "aha!" moment may come up together)

```
[ ]: IPython.display.Audio("voice/tips.mp3")
```

I just give you some small tips on the learning of algorithms. It is a sort of knowledge of knowledge. I would say, it is meta-knowledge.

They say, there are three types of thinking way. Deduction, Induction, and Abduction.

Deduction is like so-called logical thinking. If A=B and B=C, then you can conclude A=C, something like that.

In the computer process, the commands are executed one by one in order. You need to construct the calculation step by step without jumping the logic. So, the deductive thinking way is very important in programming.

Induction is a sort of generalization process. If you find some similar concepts or processes, you would be able to generalize them into one common idea or one abstract concept. There are a lot of similar processes in programming. The generalization process is very important, for example, how to group a block of code and how to define a method, and so on.

Abduction is a sort of hypothetical thinking, but not just intuition or inspiration. It is not always true but based on your experience or observation, you could get a reasonable or possible solution under some uncertainty.

Computer programming is something like a puzzle. You have a lot of pieces but you do not know how to link them to each other. Then you may do a lot of try-and-errors.

Usually, there is not only one solution, but there must be several ways to solve a problem. Sometimes you may get a new solution by abduction.

After thinking over and over again, trying a lot of combinations and possibilities, searching for knowledge in your brain, sometimes you might meet the moment. Aha or something like Bingo. It is neither deduction nor induction, but it may be an abduction.

The important thing in abduction is: not to give up thinking and to believe that the solution must exist, then you might be able to find a solution.

I will come back to this topic again from time to time during the lecture.

14 AI support tools

A paradigm shift is happening right now!!

1. **Chat type LLM tool** (ChatGPT, Claude, Gemini, Bing)
2. **AI coding agents** (Cursor, Windsurf, CLINE, VSCode + Copilot/Codeium)

15 How useful for programming?

1. Code generation, completion, suggestion (can be a collaborator/coworker)
2. Code refactoring, debugging
3. Learning (can be a teacher)

16 Discussion by ChatGPT on using ChatGPT in programming

Pros: 1. **Increased Efficiency:** Quick debugging and code completion enhance productivity. 2. **Learning Support:** Immediate resolution of beginners' questions accelerates learning. 3. **Diverse Perspectives:** Provides insights into various programming languages and up-to-date technical information.

Cons: 1. **Risk of Dependency:** Over-reliance may hinder the development of independent problem-solving skills. 2. **Risk of Misinformation:** Potential to receive incorrect information, requiring critical thinking. 3. **Creativity Inhibition:** May reduce the ability to develop unique solutions.

What do you think?

17 Prompt Engineering & Vibe Coding

17.0.1 Prompt engineering

```
is the process of designing and refining input prompts to effectively guide AI models in genera
```

17.0.2 Vibe Coding

```
is an intuitive and creative approach to coding with AI
```

17.0.3 Key points in Prompt engineering and Vibe coding

- More information leads to more specific answers
  - General questions: yield general answers
  - Specific questions: Provide context, examples, roles, input, format, etc., for more precise answers

18 Note

1. The key programming skill in the next generation will be how to use LLM tools (in my opinion)
2. Useful, but you are responsible for using it correctly (security, misinfo., personal info., trade secret)
3. You need to know the basics at least to make full use of it (at a certain point, you will be required to think by yourself)
4. Using a LLM tool in the final exam is **NOT** allowed

18.0.1 Fundamental Thinking Skills in Coding (1/2)

Thinking Skill	Self-coding (without AI)	Vibe-coding (with AI)
Deduction (Logical Construction)	Constructing logic and step-by-step structure manually	Verifying and correcting the logic in AI-generated code

Thinking Skill	Self-coding (without AI)	Vibe-coding (with AI)
Induction (Abstraction & Generalization)	Extracting patterns and generalizing into reusable components	Abstracting requirements to formulate clear prompts
Abduction (Hypothetical Reasoning)	Inferring causes of bugs or failures through trial and error	Interpreting and adjusting unexpected or unclear AI outputs

18.0.2 Structural Thinking Skills in Coding (2/2)

Thinking Skill	Self-coding (without AI)	Vibe-coding (with AI)
Framing (Problem Structuring)	Understanding assumptions and constraints of the task	Designing clear and focused prompts for AI
Goal Design (Purpose-Driven Thinking)	Implementing with a conscious awareness of the intended goal	Aligning prompt and AI output toward specific objectives
Critical Thinking (Evaluation & Judgment)	Evaluating one's own logic and implementation choices	Reviewing and selectively accepting AI-generated suggestions

19 Viewpoints of Thinking

```
<div style="text-align: center; margin: 10px;">
    <p>Self-coding:</p>
    <p style="font-size: 70%;">Engaging with each detail step by step to understand through har
    <img src="https://raw.githubusercontent.com/masaomi/bio334_2025/main/jupyter_notebooks/png/
</div>
<div style="text-align: center; margin: 10px;">
    <p>Vibe-coding:</p>
    <p style="font-size: 70%;">Seeing the whole system from above to understand structure, flow
    <img src="https://raw.githubusercontent.com/masaomi/bio334_2025/main/jupyter_notebooks/png/
</div>
```

20 My recommendation for using AI Tools in programming

For training

1. Think the solution by yourself first

2. Improve your logical thinking skills
3. Use AI tools in order to refine your thinking skills

For job tasks

1. Use AI tools to boost efficiency
2. Think independently when needed

3. Handle security and privacy with care

It will be inevitable to live together, regardless of whether it's good or bad

21 Today's Plan

1. Part1: Quick introduction, AI tools, demo, tips
2. Part2: Comparison two sequences
3. Part3: Nucleotide diversity

```
[ ]: IPython.display.Audio("voice/day1_plan.mp3")
```

Today, there are three parts.

Part1: I will just quickly review the important grammars of Python

If you think it is easy, you can skip this part.

In Part2: I just give you an example of how to compare the two nucleotide sequences

For some of you, this also may be easy part.

Then finally, in part3: You can try the implementation of the nucleotide diversity using a very simple dataset.

I just assign about one hour to each part, but you can do it by yourself at your own pace.

22 Part1 (*Quick Python Review*)

1. Basic components
2. Data structure
3. Flow control statements

Note - Please refer to Prof. CvM's group PDFs in detail - Required parts only for nucleotide diversity and Tajima's D implementation - No instruction: 1. how to use texteditor, 2. how to use terminal (Unix commands)

```
[ ]: IPython.display.Audio("voice/python_review.mp3")
```

Then let's start the quick python review.

I just pick up 3 important topics. Basic components, Data structure, and Flow control statements.

For more detail, just go back to the first module lecture or search by Google, or ask LLM services.

23 Basic Components

- Variable type (numeric, string, boolean, operand)
- Assignment operator (=)
- Numeric operators (+ - * /)
- Comparison operators (== > <)

24 Variable type (numeric string)

- Numeric: 1 2 3 1.5 3.9
- String: ‘hello’ “bye”

Note
- = assignment operator, direction right to left (variable)

[ ]:
```
IPython.display.Audio("voice/variable_type.mp3")
```

There are two types of variables. Numeric and String.

Numeric value is like 1, 2, 3,… that is, Integer value, and something like 1.5, 3.9, the real numbers.

String value is a set of letters. String value needs the single quotation or double quotation.

In computer programming, the variable is like a memory space to keep the calculation result or value. To assign a value, you use the assignment operator, =. The assignment is always from right to left. You have to put the variable on the left side of the =.

Let’s see the example below.

[ ]:
```
x = 123
print(x)
print("hello")
```

25 Numeric operators

- “+”
- “-”
- “*”
- “/”

[ ]:
```
IPython.display.Audio("voice/numeric_operators.mp3")
```

The function of the operator depends on the data type. For numeric value, addition, subtraction, multiplication, and division are available. The result becomes Integer or real number.

If you use “+” operator to Sring value, it becomes the concatenation of two strings. See the example below.

Please note one thing: The number of digits is limited in a computer. So, the result of 10 divided by 3 is rounded at the final digit and it includes a small error. See the example below.

[ ]:
```
print(1+2)
print(10/3)
print("a" + "b")
```

[ ]:
```
x = 2
y = 2
a = x + y
b = x - y
```

```
c = x * y
d = x / y

print(a,b,c,d)
```

Use round brackets if you are not sure which operator first

[ ]:
```
x = 2; y = 2;
z = (x + y)
z = (x - y)
z = (x * y)
z = (x / y)

print(a,b,c,d)
```

26 Comparison operators

```
p == q # equal
p != q # not equal
p < q  # less than
p > q  # greater than
p <= q # less than equal
p >= q # greater than equal
```

Q: What happens if it is used for String data?

[ ]:
```
IPython.display.Audio("voice/comparison_operators.mp3")
```

Additionally, there are comparison operators. These operators compare two values. Always only two values. If you want to compare three values, you have to use the comparison operator twice.

Another point in the comparison, the result becomes *True* or *False*, which is called Boolean value.

Let's see examples below.

[ ]:
```
p = 1
q = 2
print(p == q)
print(p != q)
print(p < q)
print(p > q)
print(p <= q)
print(p >= q)
```

27 Multiline comments

Use triple quotation

```
'''
all comments from here
```

```
p == q # equal
p != q # not equal
p < q  # less than
until here
'''
```

28 Multiline text (Here-document)

```
print('''
all comments from here
p == q # equal
p != q # not equal
p < q  # less than
until here
''')
```

- every line is printed out

```
[ ]: print('''
     all comments from here
     p == q # equal
     p != q # not equal
     p < q  # less than
     until here
     ''')
```

29 Boolean expression

- *True/False*
- return value of comparison operation

30 Boolean operators

- *and*: if both value are true, then *True*
- *or*: if either value is true, then *True*
- *not*: opposite of the boolean value

These are the **operators**, reserved by the system.

```
[ ]: IPython.display.Audio("voice/boolean_operators.mp3")
```

There are operators only for Boolean values, *True* and *False*.

There are three operators, *and*, *or*, *not*.

and operator returns *True* if two values are *True*.

or operator returns *True* if either value is *True*, in other words, if both values are *False*, it returns *False*.

not operator returns the opposite boolean value. *not True* becomes *False*, *not False* becomes *True*.

```
[ ]: print(True and True)
     print(True and False)
     print(True or False)
     print(False or False)
```

```
[ ]: # comparison of three values

     a = 2
     b = 5
     c = 1
     if a < b:
         if b < c:
           print("c is highest")
         else:
           print("b is highest")
     else:
         if a < c:
           print("c is highest")
         else:
           print("a is highest")
```

```
[ ]: a = 2
     b = 5
     c = 1
     if a < b and b < c:
         print("c is highest")
     elif a < b and b > c:
         print("b is highest")
     elif a > b and a > c:
         print("a is highest")
```

How do you compare more than 3 values?

31 Data structure

- *List*: a set of indexed values, mutable
- *Tuple*: a set of indexed values, immutable
- *Set*: a set of unique values, in random order
- *Dictionary*: a set of key-value data, in random order

```
[ ]: IPython.display.Audio("voice/data_structures.mp3")
```

In Python, there are four popular data structures, *List*, *Tuple*, *Set*, and *Dictionary*.

List is also called array in other programming languages. It has several elements with indexes, and the index begins with zero.

The first element of the list has the index zero. The index of the second element is 1, and the third element, 2, …, and so on.

Tuple is mostly the same as list, but it is not changeable. Once you set the elements, the element cannot change during the code running.

Set is also similar to list, but it does not have an index, and it does not allow duplicated values. Each element must be different from other elements.

Dictionary is also similar to list, but the index does not have to be an integer. Instead, it is called *key*. Namely, it is a dataset of *key* and *value*.

Let's see the example below.

32 List

- Indexed sequential data set
- Indexed from *0* (not 1)
- String object is processed like a List object

[ ]:
```
lst = [2, 4, 6, 8]
print(lst[0])
print(len(lst))

str = "hello"
print(str[0])
print(len(str))
```

Note - As you can see the example above, the String value is like a List. - But actually, it is not a List object but you can access each character of the String by index.

# Tuple * Similar to List * Immutable (you can set the values only once)

[ ]:
```
tpl = (2, 4, 6, 8)
print(tpl[0])
print(len(tpl))

tpl[0] = 100 #=> error
```

33 Set

- Unique data set in random order
- Duplicated values are removed
- Useful to remove duplicated values from a *List*
- *set()* is a built-in funciton to convert a *List* into a *Set* object

[ ]:
```
IPython.display.Audio("voice/set_and_list.mp3")
```

Set is similar to *List* but it does not have the index and it does not allow duplicated elements. The elements in the *Set* object must different from each other.

It does not have an index, which means in other words, the elements do not have order. If you need the order of elements, you may use *sorted()* method and convert it into a List object.

Let's see the example below.

[ ]:
```
s = {2, 2, 3, 3}
print(s)
string = "hello"
print(set(string))
print(sorted(set(string)))
```

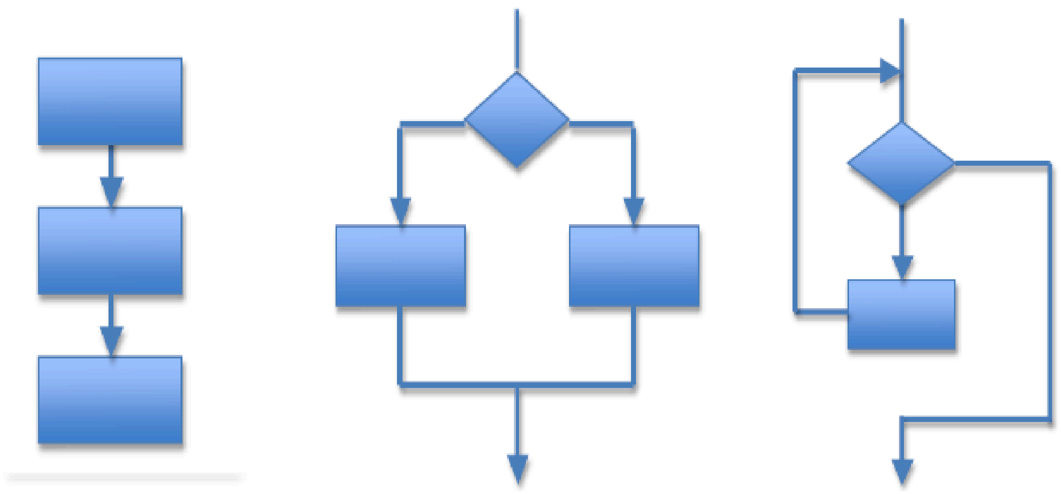
34 Dictionary

- Key-value data set in random order
- The **key** should be unique
- The **key** can be either numeric or string

[ ]:
```
d = {"apple":100, "orange":500, "banana": 300}
print(d)
print(d["apple"])
d["apple"] = 700
print(d)
print(sorted(d.keys()))
print(sorted(d.values()))
print(sorted(d.items()))
```

35 Question?

36 Flow Control



3 fundamental flows

3 fundamental flows

1. Sequential processing
2. Conditional branching
3. Repetitive processing

```
[ ]: IPython.display.Audio("voice/flow_control.mp3")
```

Next, I will explain the flow of the process. You need to remember only 3 types of process flow.

Sequential processing, conditional branching, and repetitive processing.

If you put the commands, the commands will be executed in order one by one sequentially. This is the default process flow.

Sometimes you might want to process the commands in two ways depending on the condition. Then you could use branching processing.

Sometimes, in other situations, you might want to repeat a similar process. Then you could use repetitive processing.

There are several ways to implement them. I will show you simple examples in the following.

37 Flow control statements

- *if* : conditional branching
- *while, for* : repetitive processing

38 If

conditional branching

```
if <condition>:
    # process1
else:
    # process2
```

- if **<condition>** is *True*, **process1** runs, otherwise **process2** runs

Q: How to manage multi conditions?

```
[ ]: IPython.display.Audio("voice/if_statement.mp3")
```

if-statement swithes the flow in two or more depending on a condition.

The condition should be the comparison operation or boolean operation, in other words, the condition must be *True* or *False*.

If the condition becomes *True*, the first process runs, otherwise, the second process after the *else* runs.

In other words, the condition becomes *False*, the second process runs.

Let's see the example below.

```
[ ]: x = 12
     if x > 100:
         print("process1")
     elif x > 50:
         print("process2")
     else:
         print("process3")
```

39 While

Repetitive processing

```
while <condition>:
    # process
```

- While **<condition>** is *True*, the **process** runs
- Until **<condition>** becomes *False*, the **process** runs

```
[ ]: IPython.display.Audio("voice/while_statement.mp3")
```

while-statemet is one of the statements to repeat a process. The format is simlar to if-statement.

If the condition becomes *True*, the process is repeated. In other words, the process is repeated until the condition becomes *False*. So, the condition must become *False* after a while in the repeated process somehow. Otherwise, the repeating will never end.

Let's see the example below.

```
[ ]: x = 0
     while x <= 10:
         print(x)
         x += 1

     # Q: How many times print() runs?
```

```
[ ]: # Quiz
     x = 0
     a = 0.0
     while x < 10:
         print(x)
         x += 1
         a += 0.1
     print(a == 1.0) #=> True or False?
```

40 For

Repetitive processing

```
for <variable> in <list>:
    # process
```

- Each element of **<list>** is assigned to **<variable>** in order and repeat the **process**

```
[ ]: IPython.display.Audio("voice/for_statement.mp3")
```

for-statement is another repeating statement, but a little bit more complicated and it is nice to use it with a *List* object.

Each element of the *List* is assigned to the variable and repeating the process. You can use the element information in the process but you do not have to.

The nice point of for-loop is that it is guaranteed the repeating will finish, and we can know how many times the process will be repeated in advance.

Let's see the example below.

```
[ ]: list1 = [1, 3, 5, 7]
     for i in list1:
         print(i)
```

```
[ ]: # While instead For

     list1 = [1, 3, 5, 7]
     i = 0
     while i < len(list1):
         print(list1[i])
         i += 1
```

```
[ ]: IPython.display.Audio("voice/while_and_for.mp3")
```

As you can see in the example above, the same process can be implemented either *while* or *for*, but if you repeat the process with a *List* object, the for-loop is nice and easy to handle.

Without a certain *List* object, you can generate a simple sequential *List* object by *range()* function.

Let's see other examples below.

41 For tips

A certain number of times of iteration

```
for i in range(0, n):
    print(i)
```

- The variable i is printed n times with increment

```
[ ]: for i in range(0, 4):
         print(i)
```

[ ]:
```
list1 = [1, 3, 5, 7]
for i in range(0, 4):
    print(list1[i])
```

[ ]:
```
list1 = [1, 3, 5, 7]
for i in range(0, len(list1)):
    print(list1[i])
```

[ ]:
```
# Example1: Calculate the sum of a list

list1 = [1, 2, 3, 4, 5, 6]
total = 0
for i in list1:
    total += i
print("total =", total)
```

[ ]:
```
# Example1
# Break it down when you are confused about repeat process (specialization)

list1 = [1, 2, 3, 4, 5, 6]
total = 0

total += list1[0]
total += list1[1]
total += list1[2]
total += list1[3]
total += list1[4]
total += list1[5]

print(total)
```

[ ]:
```
# Example1
# Break it down when you are confused about repeat process (specialization)

list1 = [1, 2, 3, 4, 5, 6]
total = 0

total = total + list1[0]
total = total + list1[1]
total = total + list1[2]
total = total + list1[3]
total = total + list1[4]
total = total + list1[5]

print(total)
```

[ ]:
```
# Example1
# Another way (inspiration!!)

list1 = [1, 2, 3, 4, 5, 6]
print("total =", sum(list1))

# sum() is a built-in function to calculate the sum of a list
```

[ ]:
```
# Example1
# Yet another way (inspiration!!)

print("total =", sum(range(1, 7)))

# range() is also a built-in function to generate sequential list
```

42 Mini-Summary

Generalization

- In order to process each element of *list*, use *for*

  for in : # process for each element

43 Summary

- Data type: Numeric, String, Boolean
- Data Strucutre: *List*, *Set*
- Flow Control: *if*, *while*, *for*
- **You should design an algorithm to solve a problem by using these items**
- Nucleotide Diversity and Tajima's D can be implemented using these items

Note - You can use AI, but I would suggest thinking by yourself first, otherwise you cannot improve your thinking skills in programming

[ ]:
```
IPython.display.Audio("voice/summary_day1_part1.mp3")
```

Now, you have reviewed the important Python components: Data structure and flow statements.

Let's do the exercise of the part 1 below.

But if you think it is too easy, you can skip it, and please go on to the part2.