Introduction to Python with Application in Bioinformatics

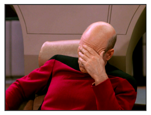
Nanjiang Shu

2024-07-15 (Day 1)

Why programming?

Typical workflow

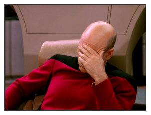
1. Get data
2. Clean, transform data in spreadsheet
3. Copy-paste, copy-paste, copy-paste
4. Run analysis & export results
5. Realise the columns were not sorted correctly
6. Go back to step 2, Repeat



Why programming?

Typical workflow

1. Get data
2. Clean, transform data in spreadsheet
3. Copy-paste, copy-paste, copy-paste
4. Run analysis & export results
5. Realise the columns were not sorted correctly
6. Go back to step 2, Repeat



With programming, you can automate some manual procesures

Why Python?

- Readability and simplicity

In [ ]:
```
# In Python
print("Hi, Python!")
```

Why Python?

- Readability and simplicity

In [ ]:
```
# In Python
print("Hi, Python!")
```

In [ ]:
```
/* In C++ */
#include <iostream>

int main() {
    std::cout << "Hi, Python!" << std::endl;
    return 0;
}
```

- Integration with C.
    - Numpy, which builds the fundatation for the popular deep learning package - Tensorflow
- `pandas` to read, manipulate and write Excel files programmtically

In [ ]:
```python
print(1 + 1)
print("Hello Python")
```

In [ ]:
```python
a = 1 + 1
print(a)
a = "Hello Python"
print(a)
```

```
In [ ]: print(1 + 1)
        print("Hello Python")
```

```
In [ ]: a = 1 + 1
        print(a)
        a = "Hello Python"
        print(a)
```

- Fixed values, e.g. `1` and `Hello Python` , in the Python code are called `literals`
- Liternals are immutable
- The name `a` that holds the value is called a `variable`

In [ ]:
```
a = 1
a = "ATCG"
a = True
a = None
```

In [ ]:
```
# Can you tell the types of them?
sequence_length = 200
scale = 2.5
gene_id = "ABC12345"
is_DNA = False
```

Use `type()` function to determine the type of a variable

In [ ]:
```
```

In [ ]:

```
sequence_length = 200
print(sequence_length)
```

In [ ]:

```
seq_len = 200
seq_lens = [100, 150, 200] # a list
print(seq_lens[1])
```

List and Tuple

- List and tuple are ordered collection of elements

```
In [ ]: seq_len = 200
        seq_lens = [100, 150, 200] # a list
        print(seq_lens[1])
```

```
In [ ]: seq_lens = (100, 150, 200) # a tuple
        print(seq_lens[1])
```

In [ ]:

```
li = [100, 150, None, "ATCG", 3.1415, seq_len, seq_lens]
li
```

Difference of List and Tuple

- List is mutable
- Tuple is immutable

In [ ]:
```
li_seqlens = [100, 150, 200]
tu_seqlens = (100, 150, 200)
```

Difference of List and Tuple

- List is mutable
- Tuple is immutable

In [ ]:

```
li_seqlens = [100, 150, 200]
tu_seqlens = (100, 150, 200)
```

In [ ]:

```
li_seqlens[1] = 500
print(li_seqlens)
```

Difference of List and Tuple

- List is mutable
- Tuple is immutable

In [ ]:
```
li_seqlens = [100, 150, 200]
tu_seqlens = (100, 150, 200)
```

In [ ]:
```
li_seqlens[1] = 500
print(li_seqlens)
```

In [ ]:
```
tu_seqlens[1] = 500
```

Set

- Set is an unordered collection of unique elements

In [ ]:
```
gene_ids = {"TP53", "COX2", "EGFR", "MTOR"} # a set
gene_ids
```

Set

- Set is an unordered collection of unique elements

```
In [ ]: gene_ids = {"TP53", "COX2", "EGFR", "MTOR"} # a set
        gene_ids
```

```
In [ ]: # set is unordered
        gene_ids = {"1", "2", "3", "4", "5"}
        for e in seq_lens:
            print(e)
```

Set

- Set is an unordered collection of unique elements

```
In [ ]: gene_ids = {"TP53", "COX2", "EGFR", "MTOR"} # a set
        gene_ids
```

```
In [ ]: # set is unordered
        gene_ids = {"1", "2", "3", "4", "5"}
        for e in seq_lens:
            print(e)
```

```
In [ ]: # set has unique element
        gene_ids = {"1", "1", "2", "2", "3"}
        print(gene_ids)
```

In [ ]:
```python
sequence_info = {  # a dictionary
    "gene": "TP53",
    "species": "Homo sapiens",
    "length": 2000
}
```

```
In [ ]: 8/2
In [ ]: 1 + 1.5
In [ ]: result = 0.1 + 0.2 - 0.3
        print(result)
```

```
In [ ]: print(result == 0.0)
```

Warning: keep in mind the precision limitations of floating-point arithmetic.

In [ ]:
```
result = 0.1 + 0.2 - 0.3
print(result)
```

In [ ]:
```
print(result == 0.0)
```

In [ ]:
```
print((result - 0.0) < 1e-6)
```

```
In [ ]: "protein"[1:4]
In [ ]: "a" == "b"
In [ ]:
```

```
In [ ]: 1 + 1.5
```

When you run operations on different data types, an underlying data type conversion has been made

```
In [ ]: 1 + 1.5
```

```
In [ ]: # Guess what will be the result for this?
        1 + True
```

What about `dkfsjdsklut`

- well, this is a valid name, but NOT recommended`

```
In [ ]: global = 5
```

```
In [ ]: # show the type of value with type()
        print(type(result))
```

```
In [ ]: # convert float value to string value with str()
        str(2.5)
```

In [ ]:
```
sequence = "ATGCTACGATaCG"
len(sequence)
```

In [ ]:
```
seq_lens = [100, 200, 300]
len(seq_lens)
```

len()

```
In [ ]: sequence = "ATGCTACGATaCG"
        len(sequence)
```

```
In [ ]: seq_lens = [100, 200, 300]
        len(seq_lens)
```

```
In [ ]: # can you get the length of an integer
        len(3)
```

```
In [ ]: read_counts = [1500, 2000, 1750, 2250, 1900, 2500]
        print("total_reads:", sum(read_counts))
```

```
In [ ]: expression_levels =  [2.5, 3.6, 4.2, 5.0, 3.8, 3.8, 9.5, 100.1]
        print("Max expression level:", max(expression_levels))
        print("Min expression level:", min(expression_levels))
```

```
In [ ]: print("Average expression level: ", sum(expression_levels)/len(expressi
In [ ]: read_counts = [1500, 2000, 1750, 2250, 1900, 2500]
        sorted_read_counts = sorted(read_counts)
        print(sorted_read_counts)
```

In [ ]:
```
seq_len1 = 150
seq_len2 = 181

seq_len1 <= seq_len2
```

In [ ]:

```
freq1 = 0.51
freq2 = 1.5

freq1 > 0.5 and freq2 > 0.5
```

```
In [ ]: gene_ids = ["TP53", "COX2", "EGFR", "MTOR"] # a set

        "TP53" in gene_ids
```

In [ ]:

```python
length = 500
species = "Mouse"
read_count = 100
# I want to evaluate the condition that length is larger than 300 or sp
# and read_count is larger than 200
# Expected value: False
length > 300 or species == "Mouse" and read_count > 200
```

A word of caution when using operators

```
In [ ]: length = 500
        species = "Mouse"
        read_count = 100
        # I want to evaluate the condition that length is larger than 300 or sp
        # and read_count is larger than 200
        # Expected value: False
        length > 300 or species == "Mouse" and read_count > 200
```

```
In [ ]: (length > 300 or species == "Mouse") and read_count > 200
```

A word of caution when using operators

In [ ]:

```python
length = 500
species = "Mouse"
read_count = 100
# I want to evaluate the condition that length is larger than 300 or sp
# and read_count is larger than 200
# Expected value: False
length > 300 or species == "Mouse" and read_count > 200
```

In [ ]:

```python
(length > 300 or species == "Mouse") and read_count > 200
```

- Always remember that **and** takes precedence over **or** in logical expressions.
- Use parentheses `()` to make your intended grouping explicit and improve readability.

```
In [ ]: mylist[1:3]
```

```
In [ ]: mylist[0:9:2] # [start, stop, step]
```

```
In [ ]: mylist[3:] # from 4th position to the end
```

```
In [ ]: mylist[:5] # from the beginning to the 5th position
```

```
In [ ]: mylist[:] # the same as mylist[::], mylist[::1]
In [ ]: mylist[-1] # return the last element, equivalent to mylist[8]
```

```
In [ ]: # What will be the result for this?
        mylist[::-1]
```

When the step is negative, it changes the direction

In [ ]:

```
mystr = "123456789"
print("mystr[2] \t= ", mystr[2] )
print("mystr[1:3] \t= ", mystr[1:3])
print("mystr[0:9:2] \t= ", mystr[0:9:2])
print("mystr[3:] \t= ", mystr[3:])
print("mystr[:5] \t= ", mystr[:5])
print("mystr[:] \t= ", mystr[:])
print("mystr[-1] \t= ", mystr[-1])
print("mystr[::-1] \t= ", mystr[::-1])
```

In [ ]:

```
# string is immutable
mystr = "123456789"
mystr[2] = "7"
```

In [ ]:

```
a = 5
print("a=", a)
a = 6
print("a=", a)
```

Are `int`, `float` and `bool` immutable?

In [ ]:
```python
a = 5
print("a=", a)
a = 6
print("a=", a)
```

In [ ]:
```python
# id(var) returns the memory address of var.
a = 5
print("memory address of a = ", id(a), ", value of a = ", a)
a = 6
print("memory address of a = ", id(a), ", value of a = ", a)
```

Are `int`, `float` and `bool` immutable?

In [ ]:
```
a = 5
print("a=", a)
a = 6
print("a=", a)
```

In [ ]:
```
# id(var) returns the memory address of var.
a = 5
print("memory address of a = ", id(a), ", value of a = ", a)
a = 6
print("memory address of a = ", id(a), ", value of a = ", a)
```

In [ ]:
```
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print("memory address of mylist = ", id(mylist), ", value of mylist = "
mylist[2] = 7
print("memory address of mylist = ", id(mylist), ", value of mylist = "
```

In [ ]:
```python
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
mylist.append(15)
print(mylist)
```

In [ ]:
```python
mylist.remove(15)
print(mylist)
```

In [ ]:
```python
del mylist[2]
print(mylist)
```

Session 3: Loops in Python

```
In [ ]: gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]
        # How do we print all IDs, one per line?
```

Session 3: Loops in Python

In [ ]:
```
gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]
# How do we print all IDs, one per line?
```

In [ ]:
```
print(gene_ids[0])
print(gene_ids[1])
print(gene_ids[2])
print(gene_ids[3])
```

Session 3: Loops in Python

In [ ]:

```
gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]
# How do we print all IDs, one per line?
```

In [ ]:

```
print(gene_ids[0])
print(gene_ids[1])
print(gene_ids[2])
print(gene_ids[3])
```

In [ ]:

```
gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

for gene_id in gene_ids:
    print(gene_id)
```

Note the INDENT of the **for** loop

Indentation is crucial in Python

- Blocks of code are defined by their indentation level.
- Typically, a tab or four spaces are used for each indentation level, but consistency within a block is the key.
- Don't mix tabs and spaces, although it's allowed.

```
In [ ]: for gene_id in gene_ids:
            print(gene_id)
```

Indentation is crucial in Python

- Blocks of code are defined by their indentation level.
- Typically, a tab or four spaces are used for each indentation level, but consistency within a block is the key.
- Don't mix tabs and spaces, although it's allowed.

```
In [ ]: for gene_id in gene_ids:
            print(gene_id)
```

```
In [ ]: for gene_id in gene_ids:
            print("==ID==")
            print(gene_id)
```

Indentation is crucial in Python

- Blocks of code are defined by their indentation level.
- Typically, a tab or four spaces are used for each indentation level, but consistency within a block is the key.
- Don't mix tabs and spaces, although it's allowed.

In [ ]:
```
for gene_id in gene_ids:
    print(gene_id)
```

In [ ]:
```
for gene_id in gene_ids:
    print("==ID==")
    print(gene_id)
```

In [ ]:
```
for gene_id in gene_ids:
        print("==ID==")
        print(gene_id)
```

Types of loops

For loop

```
In [ ]: gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

        for gene_id in gene_ids:
            print(gene_id)
```

Types of loops

`For` loop

```
In [ ]: gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

        for gene_id in gene_ids:
            print(gene_id)
```

`While` loop

```
In [ ]: gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

        i = 0
        while i < len(gene_ids):
            print(gene_ids[i])
            i += 1

        print("== When loop ends, i =", i)
```

In [ ]:
```
gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

i = 0
while i < len(gene_ids) and not gene_ids[i].startswith("E"):
    print(gene_ids[i])
    i += 1

print("== When loop ends, i =", i)
```

Examples of `while` loop

In [ ]:
```python
gene_ids = ["TP53", "COX2", "EGFR", "MTOR"]

i = 0
while i < len(gene_ids) and not gene_ids[i].startswith("E"):
    print(gene_ids[i])
    i += 1

print("== When loop ends, i =", i)
```

In [ ]:
```python
while True:
    print("yes")
```

Note: there is one built-in function called `range()` which is especially useful for the for loop

In [ ]:
```
for i in range(10):
    print(i)
```

The `range()` function

```
In [ ]: for i in range(10):
            print(i)
```

```
In [ ]: file_basename = "dnaseq"
        for i in range(10):
            seqfile = file_basename + "_" + str(i) + ".fa"
            print("Analyzing " + seqfile)
```

Session 4: Conditional statement

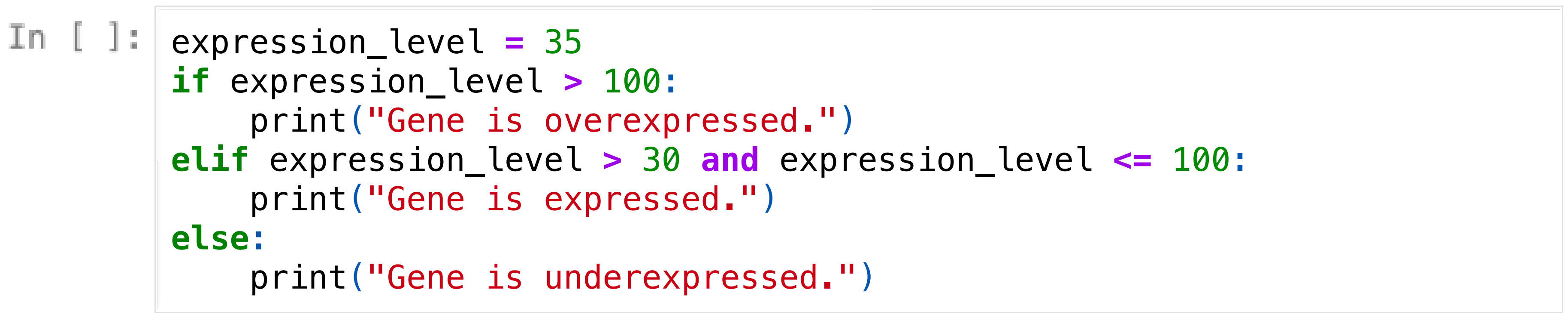
- Conditional statements allow decision-making in a program.
- Python uses `if`, `elif`, and `else` for conditionals.

In [ ]:
```
if condition1:
    # executed if condition1 is True
elif condition2:
    # executed if condition1 is False and condition2 is True
else:
    # executed if both condition1 and condition2 are False
```

In [ ]:
```
dna_sequence = "AGTCTCG"
if 'N' not in dna_sequence:
    print("Valid DNA sequence.")
```

In [ ]:

```python
expression_level = 35
if expression_level > 50:
    print("Gene is overexpressed.")
else:
    print("Gene is not overexpressed.")
```



```
In [ ]: expression_level = 35
        if expression_level > 100:
            print("Gene is overexpressed.")
        elif expression_level > 30 and expression_level <= 100:
            print("Gene is expressed.")
        else:
            print("Gene is underexpressed.")
```

In [ ]:

```python
# Use nested conditionals to categorize genetic variants based on multi
genotype = "AG"
phenotype = "expressed"
if genotype == "AG":
    # Only check phenotype if genotype is "AG"
    if phenotype == "expressed":
        print("Variant " + genotype + " is active and expressed.")
    else:
        print("Variant " + genotype + " is active but not expressed.")
else:
    print("Variant " + genotype + " is a non-target variant.")
```