Course 1

Naming rules and conventions

???????

When assigning names to objects, programmers adhere to a set of rules and conventions which help to standardize code and make it more accessible to everyone. Here are some naming rules and conventions that you should know:

- Names cannot contain spaces.
- Names may be a mixture of upper and lower case characters.
- Names can’t start with a number but may contain numbers after the first character.
- Variable names and function names should be written in snake_case, which means that all letters are lowercase and words are separated using an underscore.
- Descriptive names are better than cryptic abbreviations because they help other programmers (and you) read and interpret your code. For example, student_name is better than sn. It may feel excessive when you write it, but when you return to your code you’ll find it much easier to understand.

Common syntax errors

- Misspellings (????)
- Incorrect indentations (??????)
- Missing or incorrect key characters: (?????????)
  - Parenthetical types - (curved), [square], { curly } ???? - ??????????
  - Quote types - "straight-double" or 'straight-single', “curly-double” or ‘curly-single’ ????
  - Block introduction characters, like colons - : ??????
- Data type mismatches ????????
- Missing, incorrectly used, or misplaced Python reserved words ????????? Python ????
- Using the wrong case (uppercase/lowercase) - Python is a case-sensitive language ???????

Annotating variables by type

?????????

This has several benefits: It reduces the chance of common mistakes, helps in documenting your code for others to reuse, and allows integrated development software (IDEs) and other

tools to give you better feedback.

How to annotate a variable:

```
a = 3                #a is an integer
captain = "Picard"     # type: str
captain: str = “Picard”


import typing
# Define a variable of type str
z: str = "Hello, world!"
# Define a variable of type int
x: int = 10
# Define a variable of type float
y: float = 1.23
# Define a variable of type list
list_of_numbers: typing.List[int] = [1, 2, 3]
# Define a variable of type tuple
tuple_of_numbers: typing.Tuple[int, int, int] = (1, 2, 3)
# Define a variable of type dict
dictionary: typing.Dict[str, int] = {"key1": 1, "key2": 2}
# Define a variable of type set
set_of_numbers: typing.Set[int] = {1, 2, 3}
```

Data type conversions

Implicit vs explicit conversion ?? vs ????

Implicit conversion is where the interpreter helps us out and **automatically converts one data type into another**, without having to explicitly tell it to do so.

Example:

```
# Converting integer into a float
print(7+8.5)
```

Explicit conversion is where we **manually convert from one data type to another** by calling the **relevant function** for the data type we want to convert to.

We used this in our video example when we wanted to print a number alongside some text. Before we could do that, we needed to call the *str()* function to convert the number into a string.

- **str()** - converts a value (often numeric) to a string data type
- **int()** - converts a value (usually a float) to an integer data type
- **float()** - converts a value (usually an integer) to a float data type

Example:

```
# Convert a number into a string
base = 6
height = 3
area = (base*height)/2
print("The area of the triangle is: " + str(area))
```

Operators

Arithmetic operators

- `//` ???? (Floor division operator)
- `%` ???? (Modulo operator)
- `**` ??

Example for // & %

```
# even: □□
def is_even(number):
    if number % 2 == 0:
        return True
    return False
#This code has no ouput
```

```
def calculate_storage(filesize):
    block_size = 4096
    # Use floor division to calculate how many blocks are fully occupied
    full_blocks = filesize // block_size
    # Use the modulo operator to check whether there's any remainder
    partial_block_remainder = filesize % block_size
    # Depending on whether there's a remainder or not, return
    # the total number of bytes required to allocate enough blocks
    # to store your data.
    if partial_block_remainder > 0:
        return (full_blocks + 1) * block_size
```

```
    return full_blocks * block_size


print(calculate_storage(1))    # Should be 4096
print(calculate_storage(4096)) # Should be 4096
print(calculate_storage(4097)) # Should be 8192
print(calculate_storage(6000)) # Should be 8192
```

Comparison operators

Symbol	Name	Expression	Description
==	Equality operator	a == b	a is equal to b
!=	Not equal to operator	a != b	a is **not** equal to b
>	Greater than operator	a > b	a is larger than b
>=	Greater than or equal to operator	a >= b	a is larger than or equal to b
<	Less than operator	a < b	a is smaller than b
<=	Less than or equal to operator	a <= b	a is smaller than or equal to b

Good coding style

- **Create a reusable function** - Replace duplicate code with one reusable function to make the code easier to read and repurpose.
- **Refactor code** - Update code so that it is self-documenting and the intent of the code is clear.
- **Add comments** - Adding comments is part of creating self-documenting code. Using comments allows you to leave notes to yourself and/or other programmers to make the purpose of the code clear.
  ??????????????????????????????/????????????????????????

Loops

While Loops

```
multiplier = 1
result = multiplier * 5
while result <= 50:
    print(result)
    multiplier += 1
```

```
    result = multiplier * 5
print("Done")
```

Common errors in Loops

- **Failure to initialize variables**. Make sure all the variables used in the loop's condition are initialized before the loop.
- **Unintended infinite loops**. Make sure that the body of the loop modifies the variables used in the condition, so that the loop will eventually end for all possible values of the variables. You can often prevent an infinite loop by using the break keyword or by adding end criteria to the condition part of the *while* loop.

For Loops

```
friends = ['Taylor', 'Alex', 'Pat', 'Eli']
for friend in friends:
    print("Hi " + friend)
```

```
# °F to ℃
def to_celsius(x):
  return (x-32)*5/9


for x in range(0,101,10):
  print(x, to_celsius(x))
```

```
for number in range(1, 6+1, 2):
    print(number * 3)


# The loop should print 3, 9, 15
```

Nested for Loops

??? for ??

```
# home_team □□, away_team □□
teams = [ 'Dragons', 'Wolves', 'Pandas', 'Unicorns']
for home_team in teams:
  for away_team in teams:
    if home_team != away_team:
      print(home_team + " vs " + away_team)
```

List comprehensions

?????: `[x for x in sequence if condition]`

```
# with for loop
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x ** 2 for x in numbers]
print(squared_numbers)
```

```
# with for loop and if
sequence = range(10)
new_list = [x for x in sequence if x % 2 == 0]
```

Recursive function

???? Use cases

1. Goes through a bunch of directories in your computer and calculates how many files are contained in each.
2. Review groups in Active Directory.

```
'''
def recursive_function(parameters):
    if base_case_condition(parameters):
        return base_case_value
    recursive_function(modified_parameters)
'''
def factorial(n):
  if n < 2:
    return 1
  return n * factorial(n-1)
```

```
def factorial(n):
  print("Factorial called with " + str(n))
  if n < 2:
    print("Returning 1")
    return 1
  result = n * factorial(n-1)
  print("Returning " + str(result) + " for factorial of " + str(n))
  return result
```

```
factorial(4)
```

Types of iterables

- **String**: ??? (sequential)???? (immutable) ????????
- **List**: ??? (sequential)??? (mutable) ??????????
- **Dictionary**: ??????? key:value ??????
- **Tuple**: ??? (sequential)???? (immutable) ??????????
- **Set**: ???? (unordered)???? (unique) ???????

Resources

Naming rules and conventions

- PEP 8 – Style Guide for Python Code

Annotating variables by type

- Built-in Types — Python 3.13.0 documentation

Revision #36
Created 2 November 2024 11:24:52 by Admin
Updated 29 November 2024 13:47:33 by Admin