Programmieren I (Python)

Christian Osendorfer

2023-10-12

This weeks Friday tutorial

Python Tutor

A visual way to inspect your python programms!

Scalar object types

- `int`: integers (range of possible values of this type!)
- `float`: floating point numbers (real values)
- `bool`: booleans (`True` or `False`)
- `NoneType`: a special type with one value: `None`

Strings

Strings (“Zeichenketten”)

- Letters, special characters, spaces, digits (**not numbers**!)
- Single quoted strings and double quoted strings are equivalent:
  - `'您好, I am a string, hear me roar 🦁!'`
  - `"I've got an apostrophe"`
- Multi-line strings automatically insert new lines!

  -
    ```
    """The Zen of Python
    claims, Readability counts.
    Read more: import this."""
    # 'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
    ```
  - The `\n` is an **escape sequence** signifying a line feed.
- Concatenate strings

  -
    ```
    name = "Anna"
    hi = "Hallo"
    greeting = hi + " " + name
    ```

Strings are like lists (later!)

- A string is a sequence of characters:
  - Access the various characters through **indexing**.
    - The first position has **index 0**, the second position has **index 1**, …

  -
    ```python
    my_name = "Waldo"
    my_name[0] == 'W' # True
    ```
- But we **can not** change any character in an existing string!

  -
    ```python
    my_name = "Waldo"
    my_name[0] = 'S' # Note the useage of `=`!
    ```
  - Strings are immutable in Python!
- Many operations possible with Strings!
  - `in` operator matches *substrings*

  -
    ```python
    'Waldo' in 'Where\'s Waldo' # True
    ```

Side effects

The None value

- The special value None represents *nothingness* in Python.
- Any function that doesn’t **explicitly return a value** will return None:

  -
    ```python
    def square_it(x):
      x * x
    ```
- When a function returns None, the *console shows no output at all*:

  -
    ```python
    square_it(4)
    ```
- Attempting to treat the None like a number will result in an error:

  -
    ```python
    sixteen = square_it(4)
    sum = sixteen + 4
    ```

Side effects

- A **side effect** is when something happens as a result of calling a function besides just returning a value.
- The most common side effect is logging to the console, via the built-in **print()** function.
- Other common side effects: writing to files, drawing graphics on the screen.
- A function **without** side effecs is called **pure**, otherwise it is a **non-pure** function.

More on Functions

```
def <name>(<parameters>):
    <statement>
    <statement>
    ...
    return <return-expression>
```

Default Parameters

In the **function signature**, a parameter can specify a **default value**. If that argument *isn’t passed in*, the default value is used instead.

-
  ```python
  def calculate_dog_age(human_years, multiplier = 7):
      return human_years * multiplier
  ```
- These two lines of code have the same result:

  ```python
  calculate_dog_age(3)
  calculate_dog_age(3, 7)
  ```
- Default arguments can be overriden in two ways:

  ```python
  calculate_dog_age(3, 6)
  calculate_dog_age(3, multiplier=6)
  ```

Multiple return values (!!)

A function can specify multiple return values, separated by commas.

```python
def divide_exact(n, d):
    quotient = n // d
    remainder = n % d
    return quotient, remainder
```

- Any code that calls that function must also **unpack** these values using commas:

```python
q, r = divide_exact(618, 10)
# What's happening if you write 'q = divide_...'?
```

Doctests

Doctests check the input/output of functions. It allows *some sort* of **automated correctness testing**!

```python
def divide_exact(n, d):
    """
    >>> q, r = divide_exact(2021, 10)
    >>> q
    202
    >>> r
    1
    """
    quotient = n // d
    remainder = n % d
    return quotient, remainder
```

- How do you run a doctest?
  - `python -m doctest ...`

Boolean Expressions

Booleans

A Boolean value is either `True` or `False` and is used frequently in computer programs.

- Google Maps uses a boolean to decide whether to avoid highways in driving directions:
  - `avoid_highways = True`
- Twitter uses a boolean to remember whether the user allows personalized ads:
  - `personalized_ads = False`

Boolean Expressions

An expression can evaluate to a Boolean. Most Boolean expressions use either comparison or logical operators.

- An expression with a **comparison operator**:
  - `passed_class = grade > 65`
- An expression with a **logical operator**:
  - `wear_jacket = is_raining or is_windy`

Comparison Operators

Operator	Meaning	True expressions
`==`	Equality	`32 == 32`, `'a' == 'a'`
`!=`	Inequality	`30 != 32`, `'a' != 'b'`
`>`	Greater than	`60 > 32`
`>=`	Greater than or equal	`60 >= 32` , `32 >= 32`
`<`	Less than	`20 < 32`
`<=`	Less than or equal	`20 <= 32`, `32 <= 32`

- ⚠️ Common mistake: Do not confuse = (the assignment operator) with == (the equality operator).

Logical Operators

Operator	True expressions	Meaning
and	`4 > 0 and -2 < 0`	Evaluates to `True` if both conditions are true. If one is False evaluates to False.
or	`4 > 0 or -2 > 0`	Evaluates to `True` if either condition is true. Evaluates to False only if both are false.
not	`not (5 == 0)`	Evaluates to True if condition is false; evaluates to False if condition is true.

Compound Booleans

When combining multiple operators in a single expression, use parentheses to group:

```
may_have_mobility_issues = (age >= 0 and
age < 2)  or age > 90
```

Boolean Expressions in functions

A function can use a Boolean expression to return a result based on the values of the parameters.

```python
def passed_class(grade):
    return grade > 65

def should_wear_jacket(is_rainy, is_windy):
    return is_rainy or is_windy
```

Conditional Statement

Conditional Statement

A conditional statement gives your code a way to execute a different block of code statements based on whether certain conditions are true or false.

```
if <condition>: # This part is called a 'header line'.
    <statement> # This part is called a 'block' or a 'suite'.
    <statement>
    ...
```

A simple conditional:

```
clothing = "shirt"

if temperature < 32:
    clothing = "jacket"
```

Compound Conditionals

A conditional can include any number of `elif` statements to check other conditions.

```
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
```

For example:

```python
clothing = "shirt"

if temperature < 0:
    clothing = "snowsuit"
elif temperature < 32:
    clothing = "jacket"
```

The `else` statement

A conditional can include an `else` to specify code to execute if **no previous conditions** are `True`.

```
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
else <condition>:
    <statement>
    ...
```

For example

```
if temperature < 0:
    clothing = "snowsuit"
elif temperature < 32:
    clothing = "jacket"
else:
    clothing = "shirt"
```

Execution of conditional statements

Each clause is considered in order.

- Evaluate the header’s expression.
- If it’s `True`, execute the *suite* of statements underneath and skip the remaining clauses.
- Otherwise, continue to the next clause.

Conditionals in functions

It’s common for a conditional to be based on the value of the parameters to a function.

```python
def get_number_sign(num):
    if num < 0:
        sign = "negative"
    elif num > 0:
        sign = "positive"
    else:
        sign = "neutral"
    return sign
```

A branch of a conditional can end in a return, which exits the function entirely.

```python
def get_number_sign(num):
    if num < 0:
        return "negative"
    elif num > 0:
        return "positive"
    else:
        return "neutral"
```

`while` Statement

`While` Loops

The `while` loop syntax:

```
while <condition>:
    <statement>
    <statement>
```

As long as `condition` is `True`, the statements (a *suite*/code block) below it are executed.

```
multiplier = 1
while multiplier <= 5:
    print(9 * multiplier)
    multiplier += 1
```

Counter Variables

It’s common to use a *counter variable* whose job is keeping track of the number of iterations. Counter variables are often named as `i`, `j`, or `k`.

```
total = 0
counter = 0
while counter < 5:
  total += pow(2, 1)
  counter += 1
```

The counter variable may also be involved in the loop computation:

```
total = 0
counter = 0
while counter < 5:
  total += pow(2, counter)
  counter += 1
```

Infinite loops

```python
counter = 1
while counter < 5:
    total += pow(2, counter)
```

What one line of code would fix this?

Execution of While loops

- Evaluate the header's Boolean expression.
  - If it is a `True` value, execute the suite of statements, then return to step 1.

The `break` statement

To **prematurely** exit a loop, use the `break` statement:

```python
counter = 98
while counter < 200:
    if counter % 7 == 0:
        first_multiple = counter
        break
    counter += 1
```

The `continue` statement

To **jump back to the beginning** of the loop, use the `continue` statement:

```python
counter = 98
while counter < 200:
  counter += 1
  if counter % 7 == 0:
    first_multiple = counter
    break
  if 0 == counter % 5: # wait! Isn't this written in a funny way?
    continue
  print(counter)
```

Looping While `True`

If you are brave, you can write while loops like this:

```python
counter = 100
while True:
    if counter % 62 == 0:
        first_multiple = counter
        break
    counter += 1
```

⚠️ Be very sure that you’re not coding an infinite loop!

For Statement

For loops

The `for` loop syntax:

```
for <value> in <sequence>:
    <statement>
    <statement>
```

- ℹ️ The `for` loop has another *keyword*: `in`!

The `for` loop provides a *cleaner way* to write many `while` loops, as long as they are **iterating over some sort of sequence**.

For statement execution procedure

```
for <name> in <expression>:
    <suite>
```

- Evaluate the header `<expression>`, which must yield an **iterable value** (a **sequence**).
  - For each element in that sequence, in order:
    - *Bind* `<name>` to that element in the current frame.
    - Execute the `<suite>`.

The range type

A range represents a sequence of integers.

```
... -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5...
                range(-2, 3)
```

If having just one argument, range starts at 0 and ends just before it:

```
for num in range(6):
    print(num)       # 0, 1, 2, 3, 4, 5
```

If two arguments, range starts at first number and ends just before second:

```
for num in range(1, 6):
    print(num)       # 1, 2, 3, 4, 5
```

The break and continue statement in for loops

Both statements work exactly the same as in while loops!

```
for num in range(1, 10):
    print(num)
    if num % 2: # how is this a boolean expression ??
        print("Hi!")
        continue
    print(num*num)
    if num == 8:
      break
```

Useage of for loops

For loops are best used together with containers (**lists**, **tuples**, **dictionaries**).

```python
for j, elem in enumerate([1, 1, 2, 3, 5, 8]):
    print_me = "Element " + j + ": " + elem
    print(print_me)
```

What datatype is `[1, 1, 2, 3, 5, 8]`? What is `enumerate`?