Sterowanie procesami

Instrukcje sterujące

- iteracje - powtarzanie tej samej czynności w pętli
  - pętla nieskończona `while`
  - iteracje `for`
  - iteratory i generatory
  - biblioteka `itertools`
- warunki
  - warunek `if`
  - rozgałęzienie `if ... else`
  - rozgałęzienie `if ... elif ... else`
  - przełączanie `match ... case`
- enumeratory i słowniki

Iteracje - pętle skończone

Istotą działania pętli skończonej jest wykonanie instrukcji określoną liczbę razy. Pętle skończone w starszych językach programowania realizowane są przy pomocy zmiennej pomocniczej, której wartość jest sprawdzana za każdym przebiegiem pętli

```
for(int i=0;i<100;++i) {
  //kod programu C
}
```

W nowszych językach językach, (Python, C++) istnieją struktury iterowalne, a realizacja pętli polega na przetwarzaniu kolejno wszystkich elementów struktury

```
lista = [1,2,3,4,5]
for l in lista:
  # kod programu Python
```

```
vector<int> ar = { 1, 2, 3, 4, 5 };
for (int &x : ar)  { //kod programu C++ 17  }
```

Iteracje - pętle nieskończone

Są właściwym rodzajem pętli. Wykonywane są tak długo, aż spełniony jest warunek (`True`) zdefiniowany w nagłówku pętli, albo pętla nie zostanie przerwana instrukcją `break`. Pętle nieskończone w większości języków działają tak samo

```
while i < 100:
  #instrukcje programu
  i+=1 # jakaś forma zmiany wartości
```

instrukcje `break` i `continue` mogą być stosowane również w pętlach skończonych, jako mechanizm przerywania pętli (`break`) lub pomijania iteracji w pewnych warunkach:

```
lista = [1,3,1,7,0,4]
for l in lista:
  if l == 0:
    continue
  print(1/l)
```

Pętla nieskończona: `while True` Opuszczenie takiej pętli możliwe tylko instrukcją `break`.

Obiekty iterowalne, iteratory, generatory

Obiekty iterowalne (*iterable*) to struktury danych typu kontener, zawierające skończoną liczbę wartości, które można pobierać w kolejnych krokach iteracji (pętli skończonej). W Pythonie podstawowe struktury: listy, krotki, słowniki i zbiory są obiektami iterowalnymi.

Iterator, to struktura najczęściej wywodząca się z obiektu iterowalnego z zaimplementowaną metodą `__next__`. Większość obiektów iterowalnych można przekształcić w iterator metodą `__iter__`.

```
lista = [1,2,3,4,5]
for i in lista: # obiekt iterowalny

it = iter(lista)
next(it) # przejście do kolejnego elementu
...
rewind(it) #powrót na początek
```

Generatory to obiekty generujące kolejne elementy według zadanego wzoru, ale nie przechowujące listy wartości. Generatorem jest funkcja `range`, generująca kolejne liczby w zadanym przedziale. Iteratory i generatory to też obiekty iterowalne.

instrukcja `enumerate` i biblioteka `itertools`

`enumerate` jest instrukcją pozwalającą na pobieranie pozycji kolejnych elementów obiektu iterowalnego i samych obiektów. Jest powszechnie stosowana w procedurze iterowania po kontenerach

```
for i, color in enumerate(colors):
  # coś z kolorem oraz ze zmienną i
```

Listy składane *list comprehension* - budowane przy pomocy pętli na podstawie istniejących obiektów iterowalnych

```
[i for i in range(100) if i%2] # 1 True, 0 False
```

Moduł `itertools` jest częścią standardowej biblioteki Pythona i ostarcza funkcje do tworzenia i manipulowania obiektami iterowalnymi. Pozwala tworzyć złożone iteratory takie jak powtórzenia, kombinacje, permutacje, iloczyny kartezjańskie. Pozwala również manipulować istniejącymi iteratorami poprzez stosowanie złożonych filtrów.

https://docs.python.org/3/library/itertools.html

Rozgałęzienia

Instrukcje języka programowania, które pozwalają na wykonanie innych instrukcji w zależności od wyniku testu logicznego (czy zwraca `True` czy `False`) Możliwość decydowania kroku, jaki zostanie wykonany w dalszej kolejności jest podstawową własnością techniki obliczeniowej na jakiej opiera się komputer każdy język programowania musi ją posiadać.

Instrukcje rozgałęzień występują w kilku odmianach:

`if warunek ...` - warunkuje wykonanie kodu, w zależności do tego czy warunek jest spełniony czy nie. W przypadku niespełnienia warunku, dany fragment kodu **nie jest wykonywany**.

`if warunek ... else ...` - warunkowe wykonanie pierwszego bloku kodu, jeżeli warunek nie jest spełniony, wykonywany jest drugi blok kodu. **Jeden z dwóch bloków kodu zostanie wykonany**.

`match zmienna: case: wartość ... case _:` - znana również pod innymi nazwami (select ... case, switch ... case), zostanie wykonany jeden z fragemntów kodu, wskazany przez wartość zmiennej lub zostanie wykonany fragment kodu domyślnego (`case _:`, jeżeli został wskazany). Inaczej nic nie zostanie wykonane. Alternatywa dla mocno rozbudowanej herarchi zagnieżdzonych instrukcji `if` (Python 3.10 i wyżej)

Funkcje

Funkcje to bloki kodu, która działa jedynie wtedy gdy zostanie wywołana. Zawiera argumenty (0 lub więcej lub nieokreśloną ilość) oraz wyjście. Fukcja zwraca tylko jedną wartość

> w języku Python (i kilku innych) ograniczenie, że funkcja może zwrócić jedną wartość omija się zwracając krotkę, a więc niemodyfikowalną sekwencję dowolnych wartości, która następnie jest rozwijana do poszczególnych zmiennych.

Funkcje tworzy się albo celu wielokrotnego wykonania tego samego kodu (po to były tworzone) albo w celu izolacji fragmentów kodu (większość realnych zastosowań). W Pythonie, zmienne utworzone w funcji nie są widoczne poza funkcją (zmienne lokalne). Zmienne utworzone poza fukcją przed jej pierwszym wywołaniem są widoczne w funkcji. Zmienne przekazane do funkcji stają się lokalne.

Funkcje anonimowe (operator lambda): mikrofukcje tworzone *ad hoc* w miejscu wywołania: `lambda x: x+1`, na przykład w nagłówku innych funkcji. Stosowane najczęściej w funkcjach przyjmujących inne funkcje jako argumenty, pod warunkiem, że nie posiadają one żadnych dodatkowych arguemntów.

Uroki programowania funkcyjnego

Skondensowany przykład zasięgu zmiennych, funkcji jako argumentów i operatora lambda

```
x = 1 # zmienna globalna

def f(x,y,z): # funcja przyjmująca trzy wymagane argumenty
  return x+y+z

def g(y, f1): # funkcja przyjmująca funkcję jako argument i wykonująca dwa argumenty
  return 4 + f1(x,y)

# utworzenie funkcji anonimowej f1, która przyjmuje dwa arguemtny bo trzeci zostaje
nadany w jej ciele.
wynik = g(3,lambda x,y: f(x,y,7)) # 4 + 1 + 3 + 7 = 15
print(wynik)
```

Klasy i obiekty

Klasa jest szablonem na stworzenie obiektu. Definiuje ona właściwości oraz metody (funkcje). **Obiekt** jest _instancją _określonej klasy. Możemy stworzyć dowolną liczbę obiektów każdej klasy i każdy z nich może przechowywać inne wartości, ale ich struktura i stosowane metody są takie same.

API QGIS jest zorganizowane w klasy, których obiekty przechowują dane geoprzestrzenne. Dostęp do własności obiektów odbywa się poprzez metody, które zwracają dane, najczęściej w postaci podstawowych typów danych i prostych kontenerów. Klasy i obiekty są nakładkami na język C++.

Definicja API

Application programming interface - interface programowania aplikacji - udostępnione przez producenta aplikacji funkcje, klasy i metody pozwalające na rozbudowę narzędzia, głównie w formie wtyczek. API daje dostęp do narzędzi wykorzystywanych w działaniu aplikacji.

API składa się z trzech podstawowych elementów:

- **Protokoły –** to formaty używane do wymiany danych między poszczególnymi aplikacjami.
- **Procedury –** (routines). Odnoszą się one do konkretnych zadań albo też funkcji, które wykonują programy.
- **Narzędzia –** segmenty, z jakich można tworzyć nowe programy.

Można powiedzieć, że interfejsy programowania aplikacji działają jak pośrednik, który pozwala programistom na budowanie kolejnych funkcjonalności pomiędzy poszczególnymi aplikacjami.

Biblioteki języka Python

- biblioteka standardowa: `os`, `shutils`, `itertools`
- naukowy Python: `numpy`, `scipy`, `matplotlib`, `pandas`
- pyQt5 (aktualna wersja Qt - 6)
- `qgis` (.core i .utils)
- `processing`

Środowisko programowania skryptów

qgis.core

w języku Python API qgis jest napisane w C++ z wykorzystaniem frameworka Qt w wersji 5 (dotyczy QGIS serii 3). Dostęp do funkcji budowanych w C++ odbywa się przez bibliotekę **SIP**.

Porównanie C++ i Python:

https://api.qgis.org/api/classQgsVectorLayer.html

https://qgis.org/pyqgis/3.0/core/Vector/QgsVectorLayer.html

Qgs nie jest skrótem od Qgis ale, od Qt Gary Sherman (twóraca Quantum GIS)

API Qgs daje dostęp do pełnej funkcjonalności QGIS, zarówno w zakresie sterowania procesami obliczeniowymi jak i interface.

processing

Biblioteka processing to nakładka, napisana w Pythonie przygotowana do uruchamiania algorytmów obsługiwanych przez QGIS (zarówno natywnych jak i dostarczanych przez SAGA i GRASS). Elementami odpowiedzialnymi za uruchamianie algorytmów są klasy i metody: `QgsProcessingRegistry.createAlgorithmById()` oraz `QgsProcessingAlgorithm.run()`

```
vector = QgsVectorLayer("przekroj.gpkg")

alg_params = {
             'DISSOLVE': True,
             'DISTANCE': 100,
             'INPUT': vector,
             'OUTPUT': 'wynik.gpkg'
    }

alg_id = 'native:buffer'
buffer = QgsApplication.processingRegistry().createAlgorithmById(alg_id)
output =
buffer.run(alg_params,context=QgsProcessingContext(),feedback=QgsProcessingFeedback())

###
processing.run(alg_id,alg_params)
```

Konsola Pythona w QGIS

konsola pythona pozwala uruchamiać zarówno algorytmy geoprzetwarzania jak i daje dostęp do pełnego API. W momencie startu, niejawnie importowane są następujące klasy:

```
from qgis.core import * #1
import qgis.utils #2
import processing #3
```

1. Daje to bezpośredni dostęp do wszystkich klas z biblioteki `core`, bez konieczności importowania poszczególnych modułów
2. Daje dostęp do zmiennej `iface`, która jest instancją klasy `QgsInterface`
3. daje dostęp do funkcji biblioteki processing

W skryptach, dla wygody zmienną `iface` importuje się poleceniem `from qgis.utils import iface`. Nie zaleca się tego rozwiązania

Brak API przeznaczonego do tworzenia skryptów geoprzetwarzania

API processing pozwala jedynie na uruchamianie algorytmów geoprzetwarzania, a API Qgs jest bardzo złożone w efekcie QGIS nie posiada spójnego API przeznaczonego do tworzenia zaawansowanych skryptów geoprzetwarzania typu `ArcPy`.

Złożoność API Qgis ma swoje wady i zalety:

Zalety:

- dostęp do pełnej funkcjonalności Qgis z poziomu API
- szybkość przetwarzania właściwa dla języka C++, połączona z prostotą Pythona
- możliwość tworzenia złożonych pluginów zintegrowanych z interface

Wady:

- pomimo składni Pythona styl programowania właściwy dla C++,
- API złożone i bardzo rozbudowane, z wielpoziomowym dziedziczeniem
- mało czytelna dokumentacja, opis wielu metod w dokumentacji klas-rodziców
- proste czynności nadmiernie skomplikowane
- wymaga znajomości struktur danych przestrzennych (GDAL)

Podstawowe klasy Qgs API

- QgsProject
- QgsMapLayer
- QgsRasterLayer
- QgsVectorLayer
- QgsDataProvider
- QgsCoordinateReferenceSystem
- QgsRectangle (extent)
- QgsFeature
- QgsGeometry
- QgsField

Tworzenie warstwy

Warstwę można utworzyć na podstawie istniejącego obiektu:

```
QgsMapLayer(QgsMapLayer.VectorLayer,"path_to_layer")
QgsMapLayer(QgsMapLayer.RasterLayer,"path_to_layer")
QgsVectorLayer("path_to_layer")
QgsRasterLayer("path_to_layer")
```

Utworzona warstwa nie wczytuje danych do pamięci

QGIS Project

Klasa `QgsProject` to główna klasa obsługująca aktualny projekt. Dostęp do klasy uzyskujemy metodą `.instance()` zwracającą aktualnie otwarty projekt (prawdpopodobnie zmieni się to w QGIS 4).

Główne metody:

- `.mapLayers()` - zwraca słownik załadowanych warstw do projektu
- `.addMapLayer()` - dodaje warstwę do projektu
- `.removeMapLayer()` - usuwa trwale warstwę z projektu
- `.layerTreeRoot()` - zwraca drzewo warstw (kolejność)

`QgsProject.instance()`, przejmuje własność obiektu, tym samym jego usunięcie, trwale usuwa warstwę z projektu. Podobnie działa usunięcie warstwy z legendy.

Podstawowe klasy i metody wspólne dla obu typu danych

Metody te można wywoływać zarówno dla warstwy wektorowej jak i rastrowej. Pozwalają na pozyskanie podstawowych informacji o warstwach

- `layer.id()` - zwraca identyfikator warstwy
- `layer.name()` - zwraca nazwę warstwy
- `layer.crs()` - zwraca układ osniesienia
  - `.crs().isGeographic()` - zwraca informację czy nadana jest projekcja geodezyjna (False) czy nie (True)
  - `.crs().authid()` - zwraca nazwę bazy danych projekcji i jej identyfikator
- `layer.type()` - zwraca typ warstwy (wektor lub raster)
- `layer.source()` - źródło warstwy (np. nazwa pliku)
- `layer.extent()` - obiekt `QgsRectangle` opisujący zakres warstwy
  - `.xMinimum()`, `.xMaximum()`, `.yMinimum()`, `.xMaximum()` - odpowiednio wartości zasięgu

Wybrane metody klasy `QgsVectorLayer` (niedostępne dla rastrów)

Metody wywoływane dla warstwy wektorowej jak i obiektów wektorowych (features), pozyskanych wcześniej z warstwy wektorowej

- `vect_layer.featureCount()` - liczba obiektów
- `vect_layer.geometryType()` - zwraca typ geometrii: 0 point 1 linia 2 poligon
- `vect_layer.fields()`[0] - zwraca atrybut obiektu o indeksie 0
  - `field.name()`, `field.type()` - nazwa i typ pola
- `vect_layer.getFeatures()` - zwraca iterator obiektów warstwy wektorowej
- `vect_layer.getFeature(1)` - zwraca obiekt (feature) o indeksie 1. Obiekty są indeksowane od 1
  - `feature['LANDUSE']` - wartość pola o znanej nazwie w danym obiekcie
  - `feature[vect_layer.fields()[0].name()]` - wartość pola o znanym położeni w danym obiekcie
  - `feature.id()` - zwraca identyfikator obiektu. Obiekty są indeksowane od 1
  - `feature.geometry()` - zwraca geometrię obiektu
    - `geometry.area()` - powierzchnia obiektu
    - `geometry.length()` - długość obiektu

Wybrane metody klasy `QgsRasterLayer` (niedostępne dla warstw wektorowych)

Metody wywoływane dla warstwy rastrowej jak i obiektu dataProvider warstwy. Te ostatnie dotyczą poszczególnych pasm w pliku, gdyż typy pasm są od siebie niezależne.

`rast_layer.bandCount()` - liczba warstw w pliku
`rast_layer.width()`,`layer2.height()` - odpowiednio wysokość i szerokość
`rast_layer.dataProvider().dataType(1)` - typ danych warstwy pierwszej. Warstwy liczymy od 1
`rast_layer.dataProvider().bandStatistics(1)` - statystyki warstwy pierwszej, dostępne mn: mean, stddev, minimumValue, maximumValue,

Algoritms providers

Klasy Qgis pozwalające na zbudowanie nakładek, wywołujących natywne polecenia algorytmów QGIS (`qis`) lub innych aplikacji, w taki sposób, aby imitować algorytmy georzetwarzania QGIS, umieszczać te algorytmy w skryptach i modelach QGIS oraz ładować wyniki do środowiska aplikacji.

- **natywny (QGIS)** Dostarcza narzędzia zaimplementowane przez API QGIS oraz pluginy
- **GDAL** Dostarcza algorytmy będące nakładkami na aplikacje GDAL.
- **SAGA** Dostarcza algorytmy programu SAGA (nie wszystko działa)
- **GRASS** Dostarcza algorytmy programu GRASS GIS (większość nie działa)
- **OTB, TauDEM**

Skrypty tworzone w R

Qgis pozwala tworzyć skrypty w języku R oraz budować dla nich interface użytkownika, na innych zasadach niż skrypty tworzone w Pythonie: https://docs.qgis.org/3.34/en/docs/training_manual/processing/r_intro.html

Data Provider

Kod, który umożliwia czytanie określonych formatów danych. Najczęściej jest to GDAL/OGR obsługujący większość formatów SIG, postgis i inne formaty geoprzestrzenne, dane w pamięci operacyjnej (memory), wybrane struktury kafelkowe (vectorTile i rasterTile) chmury punktów, sceny satelitarne oraz siatki 3D (mesh), WFS (wektor) i WCS (raster)

więcej: wykład na temat GDAL/OGR.

Dostęp do narzędzi QGIS w skryptach Pytona

- Linux: `export PYTHONPATH=/<qgispath>/share/qgis/python`
- macOS: `export PYTHONPATH=/<qgispath>/Contents/Resources/python`
- Windows: `set PYTHONPATH=c:\<qgispath>\python`

```
from qgis.core import *

QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

qgs = QgsApplication([], True)
qgs.initQgis()
# kod aplikacji
qgs.exitQgis()
```