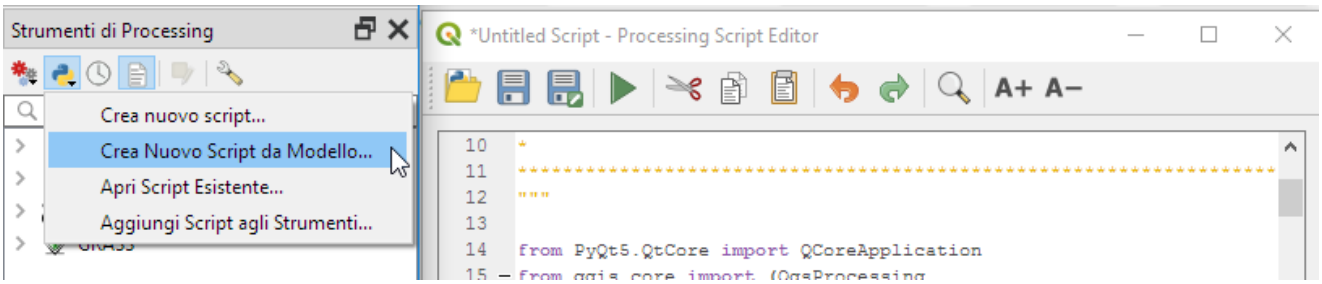
Script di processing

--

Il Framework processing può essere facilmente esteso aggiungendo degli strumenti definiti con script di python che possono usufruire di una serie di interfacce e di servizi con un minimo di codice.



[tutorial per QGIS2 - **SUPERATO**]

[template per un algoritmo di QGIS3]

[documentazione per QGIS3 - **INCOMPLETA**]

[Distribuire un plugin di processing]

[Accorgimenti per la scrittura di un plugin di processing]

Duplicazione di un Layer - QGIS2

In QGIS2 la sintassi degli script di processing è concisa e procedurale.

[05-duplicaLayers_qgis2.py]

```
##Vector=group
##input=vector
##output=output vector

from qgis.core import *
from processing.tools.vector import VectorWriter

vectorLayer = processing.getObject(input)

provider = vectorLayer.dataProvider()

writer = VectorWriter(output, None, vectorLayer.pendingFields(),
                      provider.geometryType(), vectorLayer.crs())

features = processing.features(vectorLayer)
for feat in features:
    writer.addFeature(feat)

del writer
```

--

Duplicazione di un Layer - QGIS3

In QGIS3 il framework è stato completamente riscritto in C++ per esigenze prestazionali e per consentire l'esecuzione in background. L'API è completamente cambiata ed è più complessa e poco documentata. E' da poco stato introdotto un nuovo style di scrittura degli algoritmi basato sui decoratori di Python, tramite cui, come per le espressioni è possibile aggiungere dei parametri utente di esecuzione.

06-duplicaLayers.py

```
# -*- coding: utf-8 -*-
from PyQt5.QtCore import QCoreApplication
from qgis.core import (QgsProcessing,
                       QgsFeatureSink,
                       QgsProcessingException,
                       QgsProcessingAlgorithm,
                       QgsProcessingParameterFeatureSource,
                       QgsProcessingParameterFeatureSink)
import processing


class ExampleProcessingAlgorithm(QgsProcessingAlgorithm):
```

```
    INPUT = 'INPUT'
    OUTPUT = 'OUTPUT'

    def tr(self, string):
        return QCoreApplication.translate('Processing', string)

    def createInstance(self):
        return ExampleProcessingAlgorithm()

    def name(self):
        return 'myscript'

    def displayName(self):
        return self.tr('My Script')

    def group(self):
        return self.tr('Example scripts')

    def groupId(self):
        return 'examplescripts'

    def shortHelpString(self):
        return self.tr("Example algorithm short description")

    def initAlgorithm(self, config=None):
        self.addParameter(
            QgsProcessingParameterFeatureSource(
                self.INPUT,
                self.tr('Input layer'),
                [QgsProcessing.TypeVectorAnyGeometry]
            )
        )
        self.addParameter(
            QgsProcessingParameterFeatureSink(
                self.OUTPUT,
                self.tr('Output layer')
            )
        )

    def processAlgorithm(self, parameters, context, feedback):
        """
        Here is where the processing itself takes place.
        """

        inputLayer = self.parameterAsSource(parameters, self.INPUT,
```

```
context)

        (output_sink, dest_id) = self.parameterAsSink(
            parameters,
            self.OUTPUT,
            context,
            inputLayer.fields(),
            inputLayer.wkbType(),
            inputLayer.sourceCrs()
        )

        for feat in inputLayer.getFeatures():
            output_sink.addFeature(feat)

        return {self.OUTPUT: dest_id}
```

--

Duplicazione di un Layer - QGIS3 con decoratori

A partire dalla versione 3.6 di QGIS è stata implementata una semplificazione della sintassi per ottenere uno script dall'aspetto più proceduale e conciso.

07-duplicaLayers_conDecoratori.py

```
# -*- coding: utf-8 -*-
from qgis.core import QgsProcessing
from qgis.processing import alg

@alg(name="duplica_layer", label="duplica layer", group="customscripts",
group_label=alg.tr("Custom Scripts"))
@alg.input(type=alg.SOURCE, name="INPUT", label="Input layer", types=
[QgsProcessing.TypeVectorAnyGeometry])
@alg.input(type=alg.SINK, name="OUTPUT", label="Output layer")
@alg.output(type=str, name="OUT", label="Output")
def processAlgorithm(instance, parameters, context, feedback, inputs):
    """
    Here is where the processing itself takes place.
    """

    inputLayer = instance.parameterAsSource(parameters, "INPUT", context)

    (output_sink, dest_id) = instance.parameterAsSink(
        parameters,
        "OUTPUT",
        context,
        inputLayer.fields(),
        inputLayer.wkbType(),
        inputLayer.sourceCrs()
    )

    for feat in inputLayer.getFeatures():
        output_sink.addFeature(feat)

    return {"OUTPUT": dest_id}
```

script di processing per misurare l'elevazione su un DTM lungo le polilineee QGIS2.

attenti a non definire misure <= 0

```
##dtm=raster
##input=vector
##measure=number 50
##output=output vector

from PyQt4.QtCore import QVariant
from qgis.core import *
from processing.tools.vector import VectorWriter

vectorLayer = processing.getObject(input)
dtmLayer = processing.getObject(dtm)
measureStep = measure

fields=QgsFields()
fields.append(QgsField('id_poly', QVariant.Int))
fields.append(QgsField('elevation', QVariant.Double))
fields.append(QgsField('step', QVariant.Double))

pointSamplewriter = VectorWriter(output, None, fields,
                      QgsWKBTypes.Point, vectorLayer.crs())

features = processing.features(vectorLayer)
for feat in features:
    currentLen = 0
    while currentLen < feat.geometry().length():
        point = feat.geometry().interpolate(currentLen).asPoint()
        elevFeat = QgsFeature(fields)
        elevValue = dtmLayer.dataProvider().identify(point,
QgsRaster.IdentifyFormatValue).results()[1]
        elevFeat['elevation'] = elevValue
        elevFeat['step'] = currentLen
        elevFeat['id_poly'] = feat.id()
        elevGeom = QgsGeometry.fromPoint(point)
        elevFeat.setGeometry(elevGeom)
        pointSamplewriter.addFeature(elevFeat)
        currentLen += measureStep

del pointSamplewriter
```

--

script di processing precedente modificato per il funzionamento in QGIS3

08_raster_proc_QGIS3.py

```
# -*- coding: utf-8 -*-

from PyQt5.QtCore import QCoreApplication, QVariant
from qgis.core import (QgsProcessing,
                       QgsFeatureSink,
                       QgsProcessingAlgorithm,
                       QgsProcessingParameterFeatureSource,
                       QgsProcessingParameterRasterLayer,
                       QgsProcessingParameterNumber,
                       QgsProcessingParameterFeatureSink,
                       QgsFields,QgsField,QgsFeature,QgsGeometry,
QgsWkbTypes, QgsRaster)


class DTMProcessingAlgorithm(QgsProcessingAlgorithm):
    """
    This is an example algorithm that takes a vector layer and
    creates a new identical one.

    It is meant to be used as an example of how to create your own
    algorithms and explain methods and variables used to do it. An
    algorithm like this will be available in all elements, and there
    is not need for additional work.

    All Processing algorithms should extend the QgsProcessingAlgorithm
    class.
    """

    # Constants used to refer to parameters and outputs. They will be
    # used when calling the algorithm from another algorithm, or when
    # calling from the QGIS console.

    OUTPUT_LAYER = 'OUTPUT_LAYER'
    INPUT_LAYER = 'INPUT_LAYER'
    DTM_LAYER = 'DTM_LAYER'
    MEASURE_VALUE = 'MEASURE'

    def tr(self, string):
        """
        Returns a translatable string with the self.tr() function.
        """
        return QCoreApplication.translate('Processing', string)

    def createInstance(self):
        return DTMProcessingAlgorithm()
```

```
    def name(self):
        """
        Returns the algorithm name, used for identifying the algorithm.
This
        string should be fixed for the algorithm, and must not be
localised.
        The name should be unique within each provider. Names should
contain
        lowercase alphanumeric characters only and no spaces or other
        formatting characters.
        """
        return 'measure_elev_on_dtm'

    def displayName(self):
        """
        Returns the translated algorithm name, which should be used for
any
        user-visible display of the algorithm name.
        """
        return self.tr('Measure elevation on dtm')

    def group(self):
        """
        Returns the name of the group this algorithm belongs to. This
string
        should be localised.
        """
        return self.tr('Example scripts')

    def groupId(self):
        """
        Returns the unique ID of the group this algorithm belongs to.
This
        string should be fixed for the algorithm, and must not be
localised.
        The group id should be unique within each provider. Group id
should
        contain lowercase alphanumeric characters only and no spaces or
other
        formatting characters.
        """
        return 'examplescripts'

    def shortHelpString(self):
        """
```

```
        Returns a localised short helper string for the algorithm. This
string
        should provide a basic description about what the algorithm does
and the
        parameters and outputs associated with it..
        """
        return self.tr("Example algorithm short description")

    def initAlgorithm(self, config=None):
        """
        Here we define the inputs and output of the algorithm, along
        with some other properties.
        """

        self.addParameter(QgsProcessingParameterRasterLayer(
            self.DTM_LAYER,
            self.tr("DTM layer")))
        self.addParameter(QgsProcessingParameterFeatureSource(
            self.INPUT_LAYER,
            self.tr("Input layer"),[QgsProcessing.TypeVectorLine]))
        self.addParameter(QgsProcessingParameterNumber(
            self.MEASURE_VALUE,
            self.tr("Measure step size"),
            QgsProcessingParameterNumber.Integer,50))
        self.addParameter(QgsProcessingParameterFeatureSink(
            self.OUTPUT_LAYER,
            self.tr("Output point layer"),
            QgsProcessing.TypeVectorPoint))

    def processAlgorithm(self, parameters, context, feedback):
        """
        Here is where the processing itself takes place.
        """


        vectorLayer = self.parameterAsSource(parameters,
self.INPUT_LAYER, context)
        dtmLayer = self.parameterAsRasterLayer(parameters,
self.DTM_LAYER, context)
        measureStep = self.parameterAsInt (parameters,
self.MEASURE_VALUE, context)


        fields=QgsFields()
        fields.append(QgsField('id_poly', QVariant.Int))
```

```
        fields.append(QgsField('elevation', QVariant.Double))
        fields.append(QgsField('step', QVariant.Double))


        (sink, dest_id) = self.parameterAsSink(parameters,
self.OUTPUT_LAYER, context, fields, QgsWkbTypes.Point,
vectorLayer.sourceCrs())


        features = vectorLayer.getFeatures()
#QgsProcessingUtils.getFeatures(vectorLayer, context)
        for feat in features:
            currentLen = 0
            while currentLen < feat.geometry().length():
                point = feat.geometry().interpolate(currentLen).asPoint()
                elevFeat = QgsFeature(fields)
                elevValue = dtmLayer.dataProvider().identify(point,
QgsRaster.IdentifyFormatValue).results()[1]
                elevFeat['elevation'] = elevValue
                elevFeat['step'] = currentLen
                elevFeat['id_poly'] = feat.id()
                elevGeom = QgsGeometry.fromPointXY(point)
                elevFeat.setGeometry(elevGeom)
                sink.addFeature(elevFeat, QgsFeatureSink.FastInsert)
                currentLen += measureStep

        return {self.OUTPUT_LAYER: dest_id}
```

--

script di processing precedente modificato per l'utilizzo del nuovo style con decoratori in QGIS3:

[09_raster_proc_QGIS3_decorators.py]

```
from qgis.processing import alg

from PyQt5.QtCore import QCoreApplication, QVariant
from qgis.core import (QgsProcessing,
                       QgsFeatureSink,
                       QgsFields,QgsField,QgsFeature,QgsGeometry,
QgsWkbTypes, QgsRaster)

@alg(name="measure_Elevation_along_lines", label="Measure DTM elevation
along lines", group="customscripts", group_label=alg.tr("Custom
```

```
Scripts"))
@alg.input(type=alg.MAPLAYER, name='DTM_LAYER', label="Layer DTM")
@alg.input(type=alg.SOURCE, name='INPUT_LAYER', label="Input layer",
types=[QgsProcessing.TypeVectorLine])
@alg.input(type=alg.DISTANCE, name="MEASURE_VALUE", label="Measure step
size", default=50.0)
@alg.input(type=alg.SINK, name="OUTPUT_LAYER", label="Output layer")
@alg.output(type=str, name="OUT", label="Output")
def processAlgorithm(instance, parameters, context, feedback, inputs):
    """
    Algoritm to extract elevation on DTM from linestring paths at
specified measure step
    """


    vectorLayer = instance.parameterAsSource(parameters, "INPUT_LAYER",
context)
    dtmLayer = instance.parameterAsRasterLayer(parameters, "DTM_LAYER",
context)
    measureStep = instance.parameterAsInt (parameters, "MEASURE_VALUE",
context)



    fields=QgsFields()
    fields.append(QgsField('id_poly', QVariant.Int))
    fields.append(QgsField('elevation', QVariant.Double))
    fields.append(QgsField('step', QVariant.Double))


    (sink, dest_id) = instance.parameterAsSink(parameters,
"OUTPUT_LAYER", context, fields, QgsWkbTypes.Point,
vectorLayer.sourceCrs())



    features = vectorLayer.getFeatures()
#QgsProcessingUtils.getFeatures(vectorLayer, context)
    for feat in features:
        currentLen = 0
        while currentLen < feat.geometry().length():
            point = feat.geometry().interpolate(currentLen).asPoint()
            elevFeat = QgsFeature(fields)
            elevValue = dtmLayer.dataProvider().identify(point,
QgsRaster.IdentifyFormatValue).results()[1]
            elevFeat['elevation'] = elevValue
            elevFeat['step'] = currentLen
            elevFeat['id poly'] = feat.id()
```

```
            elevGeom = QgsGeometry.fromPointXY(point)
            elevFeat.setGeometry(elevGeom)
            sink.addFeature(elevFeat, QgsFeatureSink.FastInsert)
            currentLen += measureStep

    return {"OUTPUT_LAYER": dest_id,}
```

creazione di un grafo origine/destinazione in QGIS2

10_grafo_OD_QGIS3.py

```
# -*- coding: utf-8 -*-

from PyQt5.QtCore import QCoreApplication, QVariant
from qgis.core import (QgsProcessing,
                       QgsFeatureSink,
                       QgsProcessingAlgorithm,
                       QgsProcessingParameterFeatureSource,
                       QgsProcessingParameterRasterLayer,
                       QgsProcessingParameterNumber,
                       QgsProcessingParameterFeatureSink,
                       QgsFields,QgsField,QgsFeature,QgsGeometry,
QgsWkbTypes, QgsRaster)

class GRAFOProcessingAlgorithm(QgsProcessingAlgorithm):
    """
    base algoritm class
    """

    OUTPUT_GRAFO = 'OUTPUT_GRAFO'
    OUTPUT_NODI = 'OUTPUT_NODI'
    INPUT_LINEE = 'INPUT_LINEE'

    def tr(self, string):
        """
        Returns a translatable string with the self.tr() function.
        """
        return QCoreApplication.translate('Processing', string)

    def createInstance(self):
        return GRAFOProcessingAlgorithm()

    def name(self):
```

```
    def name(self):
        return 'origine_destinazione'

    def displayName(self):
        return self.tr('Crea grafo origine/destinazione')

    def group(self):
        return self.tr('customscripts')

    def groupId(self):
        return 'customscripts'

    def shortHelpString(self):
        return self.tr("Example algorithm short description")

    def initAlgorithm(self, config=None):

        self.addParameter(QgsProcessingParameterFeatureSource(
            self.INPUT_LINEE,
            self.tr("Linee"),[QgsProcessing.TypeVectorLine]))

        self.addParameter(QgsProcessingParameterFeatureSink(
            self.OUTPUT_NODI,
            self.tr("Output nodi"),
            QgsProcessing.TypeVectorPoint))

        self.addParameter(QgsProcessingParameterFeatureSink(
            self.OUTPUT_GRAFO,
            self.tr("Output grafo"),
            QgsProcessing.TypeVectorLine))


    def processAlgorithm(self, parameters, context, feedback):
        """
        Algoritm to extract elevation on DTM from linestring paths at
specified measure step
        """

        def aggiungi_nodo(nuovo_nodo):
            for i,nodo in enumerate(lista_nodi):
                if nodo.compare(nuovo_nodo):
                    return i
            lista_nodi.append(nuovo_nodo)
            return len(lista_nodi)-1

        linee_layer = self.parameterAsSource(parameters, "INPUT_LINEE",
```

```
context)

        grafo_fields=QgsFields()
        grafo_fields.append(QgsField("rif_id", QVariant.Int))
        grafo_fields.append(QgsField("in_id", QVariant.Int))
        grafo_fields.append(QgsField("out_id", QVariant.Int))

        (grafo_sink, grafo_dest_id) = self.parameterAsSink(parameters,
"OUTPUT_GRAFO", context, grafo_fields, QgsWkbTypes.LineString,
linee_layer.sourceCrs())

        nodi_fields = QgsFields()
        nodi_fields.append(QgsField("nodo_id", QVariant.Int))
        (nodi_sink, nodi_dest_id) = self.parameterAsSink(parameters,
"OUTPUT_NODI", context, nodi_fields, QgsWkbTypes.Point,
linee_layer.sourceCrs())

        i = 0
        n = linee_layer.featureCount()
        lista_nodi = []
        feedback.pushInfo("Individuazione dei vertici degli archi del
grafo ...")

        for k,feature in enumerate(linee_layer.getFeatures()):
            feedback.setProgress(int(100*i/n))
            i += 1
            lista_vertici = feature.geometry().asPolyline()
            grafo_feature = QgsFeature()
            attributi =[feature.id()]
            grafo_feature.setGeometry(feature.geometry())
            for campo, estremo in
({1:lista_vertici[0],2:lista_vertici[-1]}).items():
                id_nodo = aggiungi_nodo(estremo)
                attributi.append(id_nodo)
            grafo_feature.setAttributes(attributi)
            grafo_sink.addFeature(grafo_feature,
QgsFeatureSink.FastInsert)

        i = 0
        n = len(lista_nodi)
        feedback.pushInfo("Creazione dei nodi ...")

        for i, nodo in enumerate(lista_nodi):
            feedback.setProgress(int(100*i/n))
            nodo_feature = QgsFeature()
            nodo_feature.setAttributes([i])
```

```
            nodo_feature.setAttributes([1])
            nodo_feature.setGeometry(QgsGeometry.fromPointXY(nodo))
            nodi_sink.addFeature(nodo_feature, QgsFeatureSink.FastInsert)

        return {"OUTPUT_NODI": nodi_dest_id, "OUTPUT_GRAFO":
grafo_dest_id}
```

--

script precedente modificato con nuovo style con decoratori:11_grafo_OD_QGIS3_decorators.py

```
from qgis.processing import alg

from PyQt5.QtCore import QCoreApplication, QVariant
from qgis.core import (QgsProcessing,
                       QgsFeatureSink,
                       QgsFields,QgsField,QgsFeature,QgsGeometry,
QgsWkbTypes, QgsRaster)

@alg(name="origine_destinazione", label="Crea grafo
origine/destinazione", group="customscripts", group_label=alg.tr("Custom
Scripts"))
@alg.input(type=alg.SOURCE, name='INPUT_LINEE', label="linee", types=
[QgsProcessing.TypeVectorLine])
@alg.input(type=alg.SINK, name='OUTPUT_NODI', label="nodi")
@alg.input(type=alg.SINK, name='OUTPUT_GRAFO', label="grafo")
@alg.output(type=str, name="OUT", label="Output")
def processAlgorithm(instance, parameters, context, feedback, inputs):
    """
    Algoritm to extract elevation on DTM from linestring paths at
specified measure step
    """

    def aggiungi_nodo(nuovo_nodo):
        for i,nodo in enumerate(lista_nodi):
            if nodo.compare(nuovo_nodo):
                return i
        lista_nodi.append(nuovo_nodo)
        return len(lista_nodi)-1

    linee_layer = instance.parameterAsSource(parameters, "INPUT_LINEE",
context)

    grafo_fields=QgsFields()
```

```
    grafo_fields.append(QgsField("rif_id", QVariant.Int))
    grafo_fields.append(QgsField("in_id", QVariant.Int))
    grafo_fields.append(QgsField("out_id", QVariant.Int))

    (grafo_sink, grafo_dest_id) = instance.parameterAsSink(parameters,
"OUTPUT_GRAFO", context, grafo_fields, QgsWkbTypes.LineString,
linee_layer.sourceCrs())

    nodi_fields = QgsFields()
    nodi_fields.append(QgsField("nodo_id", QVariant.Int))
    (nodi_sink, nodi_dest_id) = instance.parameterAsSink(parameters,
"OUTPUT_NODI", context, nodi_fields, QgsWkbTypes.Point,
linee_layer.sourceCrs())

    i = 0
    n = linee_layer.featureCount()
    lista_nodi = []
    feedback.pushInfo("Individuazione dei vertici degli archi del grafo
...")

    for k,feature in enumerate(linee_layer.getFeatures()):
        feedback.setProgress(int(100*i/n))
        i += 1
        lista_vertici = feature.geometry().asPolyline()
        grafo_feature = QgsFeature()
        attributi =[feature.id()]
        grafo_feature.setGeometry(feature.geometry())
        for campo, estremo in
({1:lista_vertici[0],2:lista_vertici[-1]}).items():
            id_nodo = aggiungi_nodo(estremo)
            attributi.append(id_nodo)
        grafo_feature.setAttributes(attributi)
        grafo_sink.addFeature(grafo_feature, QgsFeatureSink.FastInsert)

    i = 0
    n = len(lista_nodi)
    feedback.pushInfo("Creazione dei nodi ...")

    for i, nodo in enumerate(lista_nodi):
        feedback.setProgress(int(100*i/n))
        nodo_feature = QgsFeature()
        nodo_feature.setAttributes([i])
        nodo_feature.setGeometry(QgsGeometry.fromPointXY(nodo))
        nodi_sink.addFeature(nodo_feature, QgsFeatureSink.FastInsert)

    return {"OUTPUT NODI": nodi dest id, "OUTPUT GRAFO": grafo dest id}
```

I plugins di QGIS3

--

Installazione

--

- directory di installazione

```
QGIS2
[homedir]\.qgis2\python\plugins

QGIS3
[homedir]\.qgis3\python\plugins

QGIS3 - windows
[homedir]\AppData\roaming\QGIS3\python\plugins
```

- directory alternativa QGIS_PLUGINPATH in opzioni - ambiente
- installazione diretta da file zip (solo QGIS3)

struttura di un plugin

--

plugin factory a metadata.txt

La struttura minima per un plugin prevede la presenza di due files, **init**.py e metadata.txt:

init.py

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *

def classFactory(iface):
    return MinimalPlugin(iface)


class MinimalPlugin:
    def __init__(self, iface):
        self.iface = iface

    def initGui(self):
        self.action = QAction(u'Go!', self.iface.mainWindow())
        self.action.triggered.connect(self.run)
        self.iface.addToolBarIcon(self.action)

    def unload(self):
        self.iface.removeToolBarIcon(self.action)
        del self.action

    def run(self):
QMessageBox.information(None, u'Minimal plugin', u'Do something useful
here')
```

--

metadata.txt

```
[general]
name=Minimal
description=Minimal plugin
version=1.0
qgisMinimumVersion=2.0
author=Martin Dobias
email=wonder.sk@gmail.com
```

... però la tipica installazione di un plugin è più complessa e prevede la suddivisione in moduli con la separazione tra logica QGIS e logica di interfaccia QT e si troveranno tipicamente i seguenti files:

```
__init__.py               modulo di inizializzazione con plugin factory
immagini .png o .jpg      icone per l'interfaccia (button bar etc)
[modulo_principale].py    richiamato da plugin factory
[dialogo_principale].py   contiene la logica dell'interfaccia utente
[dialogo_principale].ui   contiene la definizione dell'interfaccia utente
resources.rc              talvolta contiene le risorse (immagini ed )
metadata.txt
```

---

prerequisiti per la creazione di plugin per QGIS

--

QGIS2 ed osgeo4w

Gli strumenti per la creazione di plugins per QGIS2 sono gia presenti nell'installazione base di OSGEO4W e sono inoltre impostati tutti i path correttamente in modo da far funzionare insieme tutti gli strumenti necessari:

- python2.7
- Qt4
- pyrcc4: compilatore di risorse
- pyuic4
- QT4 Designer con i QGIS widgets
- plugin builder

--

migrazione a QGIS3

la riga di comando predefinita di OSGEO4W deve essere opportunamente istruita per l'uso dei nuovi strumenti di sviluppo necessari per i plugin di QGIS3:

- eseguire lo script di configurazione di ambiente di python 3 e qt5:

```
py3_env.bat
qt5_env.bat
```

- installare i qt5-tools con il setup di osgeo4w
    - pyrcc5

- pyuic5
- QT5 Designer (QGIS widgets non compaiono)

--

Installazione dei qt5-tools con il setup di osgeo4w

--

QGIS3 in linux

UBUNTU

```
sudo apt-get install qttools5-dev-tools
```

plugin builder

--

creazione assistita

è un plugin realizzato e mantenuto da Gary Sherman fondatore del progetto QGIS per facilitare la creazione di plugin, recentemente portato a QGIS3.
Permette di creare il template di plugin a partire dalla configurazione di alcuni dati di descrizione del plugin da realizzare

--

configurazione

--

generazione

--

contenuto del plugin

contenuto del plugin appena generato

```
mioplugin                       cartella del plugin
|   icon.png                    icona per buttonbar
|   Makefile                    istruzioni per il comando make
|   metadata.txt                metadati del plugin (autore versione
etc...)
|   mio_plugin.py               classe base
|   mio_plugin_dialog.py        classe per la gestione della ui
|   mio_plugin_dialog_base.ui   file di descrizione della ui
|   pb_tool.cfg
|   plugin_upload.py            upload al repository di qgis.org
|   pylintrc
|   README.html                 informazioni sul plugin
|   README.txt                  informazioni sul plugin
|   resources.qrc               file di risorsa qt
|   __init__.py                 plugin factory
|
├──help                         directory contenente la documentazione
sphinx
├──i18n                         directory contente i files per
l'internazionalizzazione
├──scripts                      directory contente gli script di make
└──test                         directory contenente la unit test
```

--

il file resources.qrc

```
<RCC>
    <qresource prefix="/plugins/mio_plugin" >
        <file>icon.png</file>
    </qresource>
</RCC>
```

--

compilazione

http://g-sherman.github.io/Qgis-Plugin-Builder/

compilazione manuale

```
pyrcc5 -o resources.py resources.qrc
```

Queste ed altre procedure possono essere automatizzate usando make: dalle istruzioni del plugin

```
clean:         Delete the compiled UI and resource files
compile:       Compile the resource and UI files. This is the default
target.
dclean:        Same as derase but also removes any .svn entries
deploy:        Deploy the plugin
derase:        Remove the deployed plugin
doc:           Build the documentation using Sphinx
package:       Package the plugin using git archive
transclean:    Delete all .qm (translation) files
transcompile:  Compile translation files into .qm format
transup:       Update the .ts (translation) files
upload:        Upload the plugin to the QGIS repository
zip:           Deploy the plugin and create a zip file suitable for
uploading to the QGIS repository
test:          Run unit tests and produce a coverage report.
pep8:          Run python PEP8 check and produce a report.
pylint:        Run python pylint check and produce a report listing any
violations.
```

QT Designer

--

QT designer è uno strumento grafico per la definizione di interfacce utente in QT. Legge e scrive i files .ui:

--

Con QT designer possiamo editare il file .ui prodotto dal plugin builder (nell'esempio è il file mio_plugin_dialog_base.ui) ed aggiungerci dei widgets di QT. In particolare ci interessa aggiungere una finestra di testo per trasformare in plugin lo script per stampare i dettagli di un layer visto precedentemente:

--

In QT designer è inoltre possibile definire i nomi dei widgets e le loro proprietà principali

E' inoltre possibile aggiungere (in QGIS2, non ancora in QGIS3) dei custom widgets di QGIS creati appositamente per aggiungere funzionalità alle interfacce create con QT designer

--

ultime correzioni per QGIS3!! e deployment

putroppo la migrazione a QGIS3 per python non è ancora completamente automatizzata, ed è necessario andate ad editare manualmente due righe del file .ui appena modificato con QT designer per consentire a qgis di caricare correttamente le classi dei custom widgets:

aprire con un editor il file mio_plugin_dialog_base.ui

ci troveremo di fronte ad un tipico file xml e dovremo sostituire tutte le occorrenze della riga

```
<header>qgsmaplayercombobox.h</header>
```

con la seguente riga:

```
<header>qgis.gui</header>
```

a questo punto possiamo copiare la cartella del plugin nella directory che contiene i plugins di QGIS (dentro QGIS_PLUGINPATH se abbiamo configurato la variabile d'ambiente) riavviare QGIS ed attivare il plugin (il plugin sarà presente fra i plugin installati)

QT framework

--

Qt

Qt è un framework applicativo open-source sviluppato da Nokia per costruire interfacce utente grafiche (GUI) e sviluppare software. Qt è utilizzato in programmi come Google Earth, Virtual Box, Skype, Autodesk e Android. QGIS stesso è costruito con Qt. L'utilizzo di un framework applicativo come Qt velocizza il ciclo di sviluppo di un'applicazione e consente di sviluppare applicazioni multi-piattaforma.

PyQt

il modulo di collegamento (*bindings*) si chiama PyQt e può essere importato in un programma Python per controllare i widget dell'interfaccia utente
moduli di Qt
API di PyQt

--

Uso dei widgets QT

http://pyqt.sourceforge.net/Docs/PyQt5/signals_slots.html

Per programmare l'interfaccia QT, bisogna rendere sensibili gli eventi causati dall'interfaccia. Ovvero bisogna intercettare i segnali che sono scatenati da eventi predefiniti sui widgets dell'interfaccia (click per esempio) e collegarli ad uno slot, ovvero una funzione capace di gestire l'evento.

Per esempio QDialogButtonBox usato dalla nostra finestra di dialog dispone dei segnali accepted() o rejected()

nel nostro esempio il custom widget contente i layers correnti scatena un segnale layerChanged che viene connesso allo slot setLayer del custom widget contente i campi del layer scelto. una volta selezionato il campo, viene emesso il segnale fieldChanged che invierà allo slot che definiremo il nome del campo

implementazione dell interfaccia utente

--

definiamo uno slot per field changed

editare il file mio_plugin_dialog.py

```
# aggiungere l'impoartazioni dei moduli opportuni nell'instestazione
from qgis.core import  QgsWkbTypes, QgsPoint

# aggiungere ad __init__.py connessione al segnale fieldChanged

 self.mFieldComboBox.fieldChanged.connect(self.scrivi_proprieta_layer)

# aggiungere alla classe di dialogo
    def scrivi_proprieta_layer(self,nomeCampo):
        layer = self.mMapLayerComboBox.    currentLayer()
        if not layer or not layer.isValid():
          print ("Layer non valido!")
        for feature in layer.getFeatures(): #accesso alle features
            info = "'id':%d " % feature.id()
            geom = feature.geometry()
            if geom.type() ==  QgsWkbTypes.Point:
                info += "distanza da 00: %.1f " %
geom.distance(QgsPoint(0,0))
            elif geom.type() ==  QgsWkbTypes.LineString:
                info += "Lunghezza %.1f " % geom.length()
            elif geom.type() ==  QgsWkbTypes.Polygon:
                info += "Area %.1f " % geom.area()
            info += "%s: %s\n" % (nomeCampo,str(feature[nomeCampo]))
            self.plainTextEdit.appendPlainText(info)
```

alla fine questo è il risultato

--

altri strumenti utili per lo sviluppatore di python in qgis:

- plugin reloader
- first aid
- remote debug da usare insieme a winpdb o altri IDE che permettono il remote debugging (per esempio eclipse con pydev) (solo QGIS2 per ora)
- winpdb con rpdb2, introducendo nel codice opportuni segnali per bloccare il flusso di esecuzione:

esercitazioni pratiche di personalizzazione di QGIS

- modeler di QGIS: concatenare algoritmi di processing per ottenere nuovi comandi personalizzati
- realizzare un plugin di QGIS da zero
- collaborare allo sviluppo del software open source

modeler di QGIS

--

concatenare algoritmi di processing per ottenere nuovi comandi personalizzati

il "graphic modeler" o modellatore grafico è uno strumento potente di QGIS che permette di concatenare gli algoritmi di processing in cascata tra loro, usando i parametri di uscita di un algoritmo come parametri di entrata per un'altro algoritmo. Lo strumento si ispira al modelbuilder di ArcMap, di cui conserva la semplicità.In QGIS2 i modelli possono essere convertiti in script python per essere ulteriormente personalizzate. In QGIS3 la conversione non è ancora possibile ed è auspicabile che tale funzione possa essere inserita nelle prossime versioni.

E' possibile richiamare il modellatore dal menu o dal toolbox di processing.

--

Esercitazione determinare un grafico del profilo di valle

- obiettivo:
  - tracciare il profilo altimetrico del percorso più breve verso valle a partire da un dtm
- strumenti:
  - algoritmo "r.drain" di GRASS: permette di tracciare il percorso più breve verso valle a partire da un punto da specificare sullo schermo
  - algoritmo personalizzato in python creato nella scorsa lezione che permette di campionare l'elevazione su un dtm lungo una linea di input
  - algoritmo "Vector layer scatterplot": traccia un grafico XY a partire dai campi di un layer di input

--

Passi operativi. definizione dei parametri

1. scaricare il file 08_raster_proc_QGIS3.py ed installarlo come algoritmo tramite l'apposito comando dalla toolbox di processing. L'algoritmo fornisce in output un file di punti con tre attributi: id_poly: con il riferimento all'id della polilinea generatrice, elevation: elevazione in metri letta dal dtm, step: distanza dall'origine della polilinea
2. aprire il graphic modeler di processing e definire 3 parametri:
    1. "DTM" come raster layer (Layer raster)
    2. "punto di campionamento" come Point (Punto)
    3. "misura di campionamento" come Number (numero)
3. definire il nome ed il gruppo da attribuire al modello

--

--

1. passare alla scheda degli algoritmi
2. trascinare r.drain assegnando il parametro "DTM" come layer di elevazione ed il parametro "punto di campionamento" come coordinate del punto di partenza tralasciando gli altri parametri opzionali

--

- inserire l'algoritmo "measure elevation on dtm" trascinandolo dagli script utente
    - definire attribuire il layer dtm di input dal parametro dtm
    - definire il layer vettoriale di input dall'output di r.drain
    - definire la Measure step size come la variabile @misuradicampionamento: può anche essere un valore calcolato, infatti premendo sul bottone di definizione "..." appare la tipica finestra del calcolatore dei campi
    - è possibile definire l'output del modello, per esempio come "profilo di valle", in questo modo avremo un output finale supplementare. Se l'output non viene dichiarato sarà trattato come parametro temporaneo intermedio non disponibile all'utente

--

- inserire l'algoritmo "Vector layer scatterplot"
    - definire input layer dall'output di "Measure elevation on dtm"

    - inserire il nome del campo "X attribute" digitando "step": le ascisse vengono definite come distanza dal punto di campionamento
    - inserire il nome del campo "Y attribute" digitando "elevation": le ordinate vengono definite come elevazione misurata sul dtm

--

- salvare il modello. il modello finale è esportabile e può essere distribuito ad altri utenti: profilo_di_valle.model3

--

Uso del modello

il modello può essere eseguito dalla toolbox di processing sotto la voce modelli ed applicato usando il dtm già usato nelle precedenti lezioni: antelao.tif

--

Dai Modelli agli script di Processing

i modelli di processing possono venire esportati come script in python soccessivamente modificabili:

```
from qgis.core import QgsProcessing
from qgis.core import QgsProcessingAlgorithm
from qgis.core import QgsProcessingMultiStepFeedback
from qgis.core import QgsProcessingParameterRasterLayer
from qgis.core import QgsProcessingParameterNumber
from qgis.core import QgsProcessingParameterPoint
from qgis.core import QgsProcessingParameterFileDestination
import processing


class ProfiloDellaValle(QgsProcessingAlgorithm):

    def initAlgorithm(self, config=None):
        self.addParameter(QgsProcessingParameterRasterLayer('dtm', 'DTM',
defaultValue=None))

self.addParameter(QgsProcessingParameterNumber('misuradicampionamento',
'misura di campionamento', type=QgsProcessingParameterNumber.Double,
minValue=1, maxValue=1000, defaultValue=50))

self.addParameter(QgsProcessingParameterPoint('puntodicampionamento',
```

```
'punto di campionamento', defaultValue=''))
        self.addParameter(QgsProcessingParameterFileDestination('Finale',
'finale', fileFilter='File HTML (*.html)', createByDefault=True,
defaultValue=None))

    def processAlgorithm(self, parameters, context, model_feedback):
        # Use a multi-step feedback, so that individual child algorithm
progress reports are adjusted for the
        # overall progress through the model
        feedback = QgsProcessingMultiStepFeedback(2, model_feedback)
        results = {}
        outputs = {}

        # r.drain - Traces a flow through an elevation model on a raster
map.
        alg_params = {
            '-a': False,
            '-c': False,
            '-d': False,
            '-n': False,
            'GRASS_MIN_AREA_PARAMETER': 0.0001,
            'GRASS_OUTPUT_TYPE_PARAMETER': 0,
            'GRASS_RASTER_FORMAT_META': None,
            'GRASS_RASTER_FORMAT_OPT': None,
            'GRASS_REGION_CELLSIZE_PARAMETER': 0,
            'GRASS_REGION_PARAMETER': None,
            'GRASS_SNAP_TOLERANCE_PARAMETER': -1,
            'direction': '',
            'input': parameters['dtm'],
            'start_coordinates': parameters['puntodicampionamento'],
            'start_points': None,
            'drain': QgsProcessing.TEMPORARY_OUTPUT,
            'output': QgsProcessing.TEMPORARY_OUTPUT
        }
        outputs['RdrainTracesAFlowThroughAnElevationModelOnARasterMap'] =
processing.run('grass7:r.drain', alg_params, context=context,
feedback=feedback, is_child_algorithm=True)

        feedback.setCurrentStep(1)
        if feedback.isCanceled():
            return {}

        return results

    def name(self):
        return 'profilo della valle'
```

```
    def displayName(self):
        return 'profilo della valle'

    def group(self):
        return 'mio'

    def groupId(self):
        return 'mio'

    def createInstance(self):
        return ProfiloDellaValle()
```

Modelli e Script di Processing possono essere importati dal repository di QGIS

un nuovo plugin di QGIS

--

plugin di estrazione altimetrica dal sito del ministero dell'ambiente

Un classico campo di utilizzo dei plugin di QGIS è l'accesso a informazioni remote tramite internet. l'obiettivo dell'esercitazione è creare un nuovo plugin di QGIS3 per determinare l'elevazione di un punto sullo schermo per mezzo del servizio cartografico del ministero dell'ambiente.

La creazione del plugin si articola nelle seguenti attività:

- creazione della struttura del plugin tramite plugin builder
- disegno dell'interfaccia utente
- compilazione delle risorse ed installazione in QGIS
- creazione di una procedure di interrogazione al server del ministero dell'ambiente
- creazione della procedure per intercettare il click sullo schermo
- logica per il trattamento dei risultati ricevuti

--

Plugin builder

1. assicurarsi di aver opportunamente configurato la variabile QGIS_PLUGINPATH su una directory locale di sviluppo, per esempio documenti[qgis_dev]
2. aprire il plugin builder, usiamo il nome "altimetria" come nome del plugin e author/email a piacere
3. inserire un breve "about", sarà inserito nei metadati
4. selezionare "tool button with dock widget" come template ed altimetria come testo per il nuovo menu
5. deselezionare le altre opzioni (internationalization, help unit test ...)
6. selezionare la directory indicata in QGIS_PLUGINPATH come directory di output

--

--

Modifica del file di interfaccia utente .ui con QT Designer

dovremo configurare tre widget:

1. aprire il file altimetria_dockwidget_base.ui
2. trascinare una riga di testo per i dettagli sul punto di campionamento (latitudine, longitudine, sistema di riferimento). denominare l'oggetto come "posizione"
3. trascinate una riga di testo che conterra l'altimetria estratta. denominare l'oggetto come "elevazione"
4. trascinare un bottone per attivare la modalità di campionamento su schermo. denominare l'oggetto come "attivazione" e modificare l'etichetta sul bottone con un doppio click

--

Compilazione del file delle risorse ed installazione del plugin

1. aprire una finestra di comando di osgeo
2. configurare l'ambiente per python3/qt5:

```
py3_env.bat
qt5_env.bat
```

3. compilare il file con il comando pyrcc5:

```
pyrcc5 -o [homepath]/documenti/qgis_dev/altimetria/resources.py
[homepath]/documenti/qgis_dev/altimetria/resources.qrc
```

4. riavviare QGIS ed attivare il plugin dalla finestra di dialogo del plugins
5. comparirà un bottone nella toolbar (con la classica icona predefinita se non opportunamente cambiata) premendo il quale fa comparire in basso a sinistra il dockwidget che abbiamo definito in QT designer

--

interrogazione del geoportale del Ministerto dell'ambiente

il geoportale nazionale ci permettere di osservare il DTM con risoluzione a 20m dal seguente indirizzo: http://www.pcn.minambiente.it/viewer/index.php?services=dtm_20m

Esplorando con gli strumenti di sviluppo le chiamate di rete è possibile identificare la chiamata http che permettere di ricevere l'informazione che ci interessa:

```
http://www.pcn.minambiente.it/arcgis/rest/services/dtm/dtm_20m/MapServer/identify?f=json&geometry={"x":12.27250,"y":46.46501,"spatialReference":{"wkid":4326}}&tolerance=2&returnGeometry=true&mapExtent={"xmin":-2011404.6228092457,"ymin":4032867.6657353314,"xmax":2608229.616459233,"ymax":5449714.249428499,"spatialReference":{"wkid":32633}}&imageDisplay=1940,595,96&geometryType=esriGeometryPoint&sr=32633&layers=all:0
```

la chiamata restituisce la seguente risposta:

```
{"results":[{"layerId":0,"layerName":"DTM 20
m","value":"2191","displayFieldName":null,"attributes":{"Stretched
value":"255","Pixel
Value":"2191","objectid":"2192","count":"40252"},"geometryType":"esriGeom
etryPoint","geometry":
{"x":290583.51147655566,"y":5149330.284960947,"spatialReference":
{"wkid":32633}}}]}
```


--

procedure di interrogazione tramite il modulo request

Per interrogare il geoportale verrà utilizzato il modulo requests che permette di effettuare una richesta http, ricevere ed analizzarne la risposta direttamente dal codice python:

```
import requests

def ottieni_elevazione(campionamento,contesto,sr): #QgsPoint,
QgsRectangle, QgsCoordinateReferenceSystem
    srid = sr.postgisSrid()
    url_req =
'http://www.pcn.minambiente.it/arcgis/rest/services/dtm/dtm_20m/MapServer
/identify'
    parametri ={
        'f':'json',
        'geometry': '{"x":%f,"y":%f,"spatialReference":{"wkid":%d}}' %
(campionamento.x(),campionamento.y(),srid),
        'tolerance': 2,
        'imageDisplay': '1940,595,96',
        'geometryType': 'esriGeometryPoint',
        'layers': 'all:0',
        'sr': srid,
        'mapExtent':
'{"xmin":%f,"ymin":%f,"xmax":%f,"ymax":%f,"spatialReference":
{"wkid":%d}}' %
(contesto.xMinimum(),contesto.yMinimum(),contesto.xMaximum(),contesto.yMa
ximum(),srid)
    }
    r = requests.get(url_req, params=parametri)
    if r.status_code == 200:
        risposta = r.json()
        try:
            return risposta['results'][0]['value']
        except:
            return None
    else:
        return None
```

--

verifica

la procedura di interrogazione può essere verificata creando uno script utente python:

1. creare un nuovo script utente nella console di python
2. eseguire il codice, si ottiene così la nuova funzione "ottieni_elevazione"
3. determinarsi dei parametri di test dalla finestra di visualizzazione corrente

```
>>> schermo = iface.mapCanvas().extent()
>>> centro = schermo.center()
>>> srif = iface.mapCanvas().mapSettings().destinationCrs()
```

4. eseguire quindi la funzione con i parametri ricavati

```
>>> ottieni_elevazione(centro,schermo,srif)
```

--

Implementazione del plugin "altimetria"

bisogna adesso passare all'implementazione dei comportamenti previsti per il plugin. Il flusso di operazioni previsto è:

1. click sul bottone per attivare lo strumento di lettura delle coordinate di puntamento
2. campionamento delle coordinate ottenute tramite la funzione "ottieni_elevazione" appena creata
3. scrittura del risultato sulle finestre di testo

La classe che permette di intercettare un evento sullo schermo e: QgsMapTool. La classe restituisce ai metodi canvasPressEvent, canvasMoveEvent, canvasReleaseEvent e canvasDoubleClickEvent i dettagli sull'evento intercettato (per esempio le coordinate schermo)
Dovremo quindi definire una classe personalizzata che ci restituirà la posizione selezionata dall'utente

--

modifica del file altimetria_dockwidget.py

aggiungere la classe che eredita *QgsMapTool*
quando viene cliccato lo schermo viene chiamata la funzione canvasReleaseEvent, lo strumento comunica la posizione e l'elevazione tramite il segnale *catturaElevazione*

```
class intercetta(QgsMapTool):

    catturaElevazione = pyqtSignal(QgsPointXY,float)

    def __init__(self,iface):
        self.iface = iface
        super(intercetta, self).__init__(iface.mapCanvas())

    def canvasDoubleClickEvent(self, event):
        puntoDoppioClick = event.mapPoint()
        schermo = self.iface.mapCanvas().extent()
        srif = self.iface.mapCanvas().mapSettings().destinationCrs()
        elevazione = ottieni_elevazione(puntoDoppioClick,schermo,srif)
        if elevazione:

 self.catturaElevazione.emit(puntoDoppioClick,float(elevazione))
```

aggiungere in testa al file la funzione *ottieni_elevazione* precedentemente definita in modo da disporre direttamente della funzionalità di interrogazione remota.

--

modifica della funzione init

intercettare il click sul pulsante di abilitazione ed ottenere il riferimento a QgisInterface

```
    def __init__(self,iface, parent=None):
        """Constructor."""
        super(altimetriaDockWidget, self).__init__(parent)
        self.setupUi(self)
        self.iface = iface
        self.strumentoIntercetta = intercetta(iface)
        self.attiva.clicked.connect(self.intercetta_dbClick)
        self.strumentoIntercetta.catturaElevazione.connect(self.aggiorna)

    def intercetta_dbClick(self):
        self.iface.mapCanvas().setMapTool(self.strumentoIntercetta)

    def aggiorna(self,punto,elevazione):
        print ("aggiorna", elevazione)
        self.posizione.setText("%f,%f" % (punto.x(),punto.y()))
        self.elevazione.setText(str(elevazione))
```

modificare il metodo costruttore nella riga 227 file altimetria.py

```
self.dockwidget = altimetriaDockWidget(self.iface)
```

plugin installabile

collaborare allo sviluppo del software open source

--

la piattaforma github

Github (*Ghi-tab* per i non italiani) è una piattaforma web che permette agli sviluppatori di conservare e pubblicare i codici sorgenti dei propri programmi, senza perdere traccia delle versioni del codice stesso. Nel contempo permette a gruppi di sviluppatori di lavorare simultaneamente sullo stesso progetto evitando conflitti.
E' basato su Git, il potente strumento di versionamento creato da Linus Torvarlds, il creatore di Linux, per gestire lo sviluppo collaborativo del sistema operativo open source.

- Git è uno strumento a linea di comando, la cui comprensione esula dagli obiettivi del corso. Per approfondimenti si può fare riferimento ad uno dei tanti tutorial presenti in internet: https://www.slideshare.net/stefanovalle/guida-git
- E' ottimizzato per tenere traccia delle modifiche di file testo a livello di riga (non va bene quindi per file binari o per file di testo con poche righe)

--

Git

per facilitare l'interazione dell'utente, github mette a disposizione Github desktop (disponibile per MacOS e Windows), che permette di gestire con facilità i vari flussi di lavoro senza necessariamente conoscere Git. E' comunque importante comprendere i principi di funzionamento di Git:

--

Lessico di git/github

- *repository*: archivio di files e directory gestito da git. può essere locale o remoto

- *commit*: insieme coordinato di modifiche che l'utente registra sul repository
- *branch*: uno stato del repository che viene memorizzato dall'utente separatamente da altri branch. Esistono dei branch di sistema (master, origin etc...) e dei branch utente
- *diff*: operazione che mette in evidenza le modifiche di riga tra branch
- *merge*: operazione che permette di fondere tra loro due branch (per esempio un branch di sviluppo nel branch master) mettendo in evidenza eventuali conflitti
- *clone*: operazioone di clonazione in locale di un repository remoto
- *push*: operazione con la quale si si conferisce (submit) un branch locale ad un repository remoto tenendo condo dei conflitti tra versioni
- *pull*: operazione con la quale si scarica in locale un repository remoto
- *pull request*: operazione con la quale un utente propone la modifica di un repository

--

github desktop

E' fondamentalmente un'interfaccia grafica di Git integrata con il servizio di Github

--

Esercitazione di Github

- accreditarsi su Github, scaricare ed installare Github desktop
- inizializzare un repository
- clonare un repository
- modificare i files / verificare le differenze / realizzare un commit
- creare un branch
- fondere (merge) due branch
- pubblicare un repository
- inviare una pull request (PR)